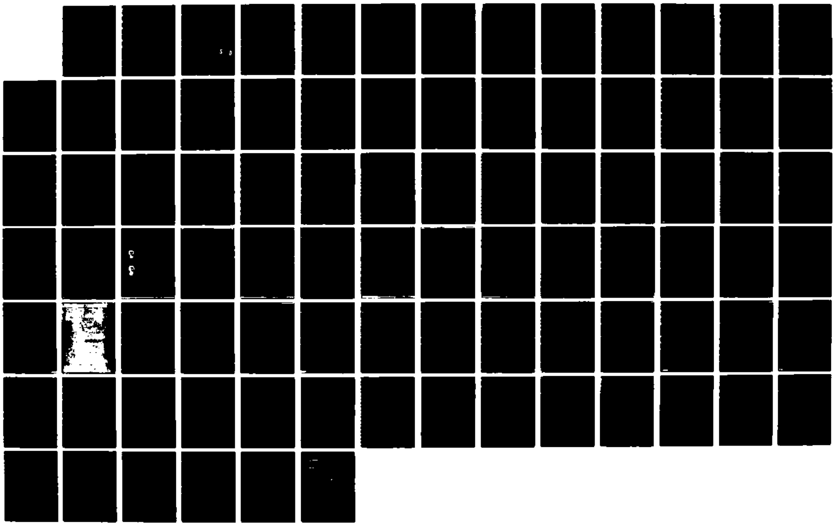
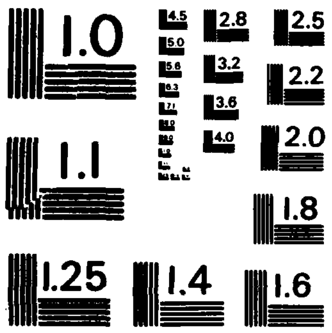
AD-A166 081   A COMPUTABILITY THEORY FOR DISTRIBUTED SYSTEMS(U) TEXAS   1/1
UNIV AT AUSTIN DEPT OF COMPUTER SCIENCES J MISRA
13 MAR 86 N00014-85-K-0057

UNCLASSIFIED   F/G 9/2   NN




1.0
4.5
5.0
5.6
6.3
7.1
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

12

Final Technical Report

Navy Grant N00014-85-K-0057

Office of Naval Research

A Computability Theory for Distributed Systen

DEPARTMENT OF COMPUTER SCIENCES
THE UNIVERSITY OF TEXAS AT AUSTIN

AUSTIN, TEXAS 78712

12

Final Technical Report

Navy Grant N00014-85-K-0057

Office of Naval Research

A Computability Theory for Distributed Systems

Principal Investigator: Jayadev Misra

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

March 13, 1986

Final Report: N00014-85-K-0057 [illegible]

Summary of Work Accomplished

The work proposed under this grant was to develop theories which will contribute to the design and analysis of distributed systems. The major emphasis of the proposed research was to study how and why processes in a message passing system need to communicate. The research results have far exceeded our initial expectations: in the following pages we describe the summary of the work performed and what these results imply for the development of distributed systems. There have been two journal papers (Distributed Computing, Springer-Verlag), one conference paper (Symposium on Principles of Programming Languages '86), and several technical reports published under this grant.

How Processes Learn [4,1]

Processes in distributed systems communicate with one another exclusively by sending and receiving messages. A process has access to its state but not to the states of other processes. Many distributed algorithms require that a process determine facts about the overall system computation. In anthropomorphic terms, processes "learn" about states of other processes in the evolution of system computation. This work is concerned with how processes learn. We give a precise characterization of the minimum information flow necessary for a process to determine specific facts about the system.

The central concept in our study is that of *isomorphism* between system computations with respect to a process: two system computations are isomorphic with respect to a process if the process behavior is identical in both. In anthropomorphic terms, "system computations are isomorphic with respect to a process," means the process cannot distinguish between them.

Many correctness arguments about distributed systems have the following operational flavor: "I will send a message to you and then you will think that I am busy and so you will broadcast ...". Such operational arguments are difficult to understand and error prone. The basis for such operational arguments is usually a "process chain": a sequence of message transfers along a chain of processes. We advocate nonoperational reasoning. The basis for nonoperational arguments is isomorphism; we relate isomorphism to process chains. Algebraic properties of system computations under isomorphism provides a precise framework for correctness arguments.

It has been proposed that a notion of "knowledge" is useful in studying distributed computations. In earlier works, knowledge is introduced via a set of axioms. Our definition of knowledge is based on isomorphism. Our model allows us to study how knowledge is "gained" or "lost". One of our key theorems states that knowledge gain and knowledge loss both require sequential transfer of information: if process q does not know fact b and later, p knows that q knows b, then q must have communicated with p, perhaps indirectly through other processes, between these two points in the computation; conversely, if p knows that q knows b and later, q does not know b then p must have communicated with q between these two points in the computation. In the first case, the effect of communication is to inform

p of q's knowledge of b. Analogously, in the second case, the effect of communication is to inform q of p's intention of relinquishing its knowledge (that q knows b). Generalizations of these results for arbitrary sequences of processes are stated and proved as corollaries of a general theorem on isomorphism.

We use the results alluded to in the last paragraph for proving lower bounds on the number of messages required to solve certain problems. We show, for instance, that there is no algorithm to detect termination of an underlying computation using only a bounded number of overhead messages.

We have extended this work in [1] to deduce facts from incomplete information.

Reaching Agreement in Faulty Distributed Systems [2,3]

Reaching agreement in a faulty distributed system, also known as Byzantine agreement, has been a central problem in fault-tolerant distributed computing. Our interest in studying this problem was to develop theories and conditions for fault tolerance in various different situations. We studied an important algorithm due to Fischer, Lynch and Fowler ("A Simple and Efficient Byzantine Generals Algorithm," *Proceedings of the 2nd Symposium on Reliability in Distributed Software and Database Systems*, July, 1982) and proposed a proof of it along traditional lines of program proving. This work has resulted in a very compact version of this algorithm. Another important result, again due to Fischer and Lynch, shows that agreement requirements synchrony: in an asynchronous system, agreement on a common value cannot be reached even if only one process fails. We provided a proof of this result using a set of simple axioms. Our proof also includes a number of key lemmas and clarifies the relationship between agreement and decision making by a process.

An Abstract Concurrent Model and Its Temporal Logic [5]

{The work of Professor Pnueli was supported by this grant}

We advance the radical notion that a computational model based on the *reals* provides a more abstract description of concurrent and reactive systems, than the conventional *integers* based behavioral model of execution *sequences*. This model is studied in the setting of temporal logic, and we illustrate its advantages by providing a *fully abstract* temporal semantics for two simple concurrent languages, and examples of specification and verification of concurrent programs within the real temporal logic defined here. It is shown that, by imposing the crucial condition of *finite variability*, we achieve a balanced formalism that is insensitive to *finite* stuttering, but can recognize *infinite* stuttering, a distinction which is essential for obtaining a fully abstract semantics of nonterminating processes. Among other advantages, going into real-based semantics obviates the need for the controversial representation of concurrency by interleaving, and most of the associated fairness constraints.

Systolic Arrays as Programs [6]

Systolic algorithms represent a form of parallel programming in which a number of nodes (machines) are interconnected by a set of lines. A node reads from its input lines and writes into its output lines on specific clock pulses and there are only a few kinds of nodes each doing different kinds of processing. Systolic algorithms are typically described by pictures of the interconnections of nodes, descriptions of processing at each node in the picture and data movements among nodes at several successive steps. A pictorial representation guarantees that the algorithm can be implemented on a VLSI chip without wire crossings, for instance. However, a picture makes it difficult to argue about the correctness of the algorithm, explore alternate designs or even develop an algorithm in a systematic manner.

We propose to view systolic algorithms as programs and hence, apply traditional program development techniques, based on invariants, in their design. We carry out the developments for matrix multiplication of band matrices and L-U decomposition of a band matrix.

Technical Reports

(1) "Learning from Incomplete Information," Technical Report, Department of Computer Sciences, University of Texas, September 1985, (K. Mani Chandy and Jayadev Misra).

(2) "Understanding a Byzantine Algorithm," TR-85-20, Department of Computer Sciences, University of Texas, September 1985, (Jayadev Misra).

(3) "On the Non-existence of Robust Commit Protocol," Technical Report, Department of Computer Science, University of Texas, November 1985, (K. Mani Chandy and Jayadev Misra).

Publications

(4) "How Processes Learn," *Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing*, Minaki, Ontario, Canada, August 5-7, 1985. Also appeared in *Journal of Distributed Computing*, Vol. 1, No. 1, pp. 40-52 (Springer-Verlag), (K. Mani Chandy and Jayadev Misra).

(5) "A Really Abstract Concurrent Model and its Temporal Logic," *Proceedings of the Thirteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, St. Petersburg, Florida, January 13-15, 1986, pp. 173-183, (Howard Barringer, Ruurd Kuiper and Amir Pnueli).

(6) "Systolic Algorithms as Programs," to appear in the special issue on VLSI, *Journal of Distributed Computing*, Vol. 1, No. 3, (K. Mani Chandy and Jayadev Misra).

Learning from Incomplete Information

(Extended Abstract)

K. Mani Chandy
Jayadev Misra

Department of Computer Sciences
The University of Texas
Austin, Texas 78712
(512)471-4353

12 September 1985

This work was supported by a grant from the Office of Naval Research under grant number ONR N00014-85-K-0057.

"When you have eliminated the impossible, whatever remains, however improbable, must be the truth."[1]

Introduction

Deduction, according to Holmes, is based upon: (1) plausibilities, (2) clues, and (3) elimination of impossibilities given the clues. Holmes starts his detection with a set of possible scenarios, i.e., a model of the crime. Then he gathers clues and eliminates scenarios that are incompatible with his clues. Our view of knowledge is similar. We start with a model (set of possible scenarios), we make certain observations of the system and thereby, we eliminate scenarios that are incompatible with the observations. In this paper, we are concerned exclusively with deduction, i.e., elimination of scenarios incompatible with observations. We do not address the question of how the model is postulated in the first place, nor with methods for making observations.

For a variety of reasons, one cannot always observe everything one wants to. Deductions are necessary precisely because observations are often incomplete. For example, reasons for an aircraft crash has to be deduced from the information in voice and data recorders, control tower recordings and memories of survivors. Software is debugged by collecting values of certain key variables over some points in execution and this partial observation may enable a programmer to deduce that the program has an error, i.e., program behavior is incompatible with a correct implementation. Fault detection and location in electronic circuits are often carried out by observing the voltage levels on some specified lines; it is usually too expensive to probe all lines. A design task is to identify the lines that may be probed and make those lines accessible. This is akin to deciding, at the design stage, what must be observable to gain certain kinds of knowledge. We propose a model of computation which captures the essence of partial observations.

Given a partial observation, all scenarios that could have produced that observation are isomorphic with respect to that observation; all other scenarios can be eliminated from further consideration. In general, if more is observed, more scenarios can be eliminated and hence more knowledge about the system can be gained.

We propose a general model of discrete systems in terms of events and relationships among them. Our model appears to be general enough to encompass all known distributed systems including the usual message passing and shared variable models. We define the notion of isomorphism among computations with respect to a process, i.e., two computations are isomorphic with respect to a process, if the process cannot

[1]Sherlock Holmes in *The Sign of the Four*, [Chapter 6], by Sir Arthur Conan Doyle, [1890].

distinguish between the two computations. Our fundamental theorem relates isomorphism to communications among processes in a computation. We define knowledge using isomorphism and show the types of communications needed to gain or lose knowledge. We postulate that some pairs of events on a process may not be distinguishable to it. Thus, a process may know that a certain type of event has occurred, but it may not be able to say which particular event of that type has occurred. This is a mechanism for capturing the notion of partial observation.

Model

A system is a set of events E and two binary relations $\rightarrow$ and $\sim$ on events, where $\rightarrow$ is a partial order and $\sim$ is irreflexive and symmetric. A computation is any sequence of events x satisfying the following two conditions. For any e,e' in E:

1. **precedence:** if $e \rightarrow e'$ and e' is in x then e occurs prior to e' in x.

2. **exclusivity:** if $e \sim e'$ than at most one of e,e' is in x.

An event represents a discrete action; $e \rightarrow e'$ means that event e' can occur only after event e has occurred; $e \sim e'$ means that both e,e' cannot occur in the same computation. Note that, if $e \rightarrow e'$ and $e \sim e'$ then e' can occur in no computation.

Our definition of system is given independent of the processes at which an event may take place. A number of important properties of computations, such as prefix closure etc., may be proven from our definition. However, an adequate theory of knowledge requires us to postulate processes which are not omniscient. We do so next.

Each process has a set of invisible events; these are the ones in which it presumably does not participate and whose occurrences it cannot observe; remaining events are visible to it. Furthermore, a process may be unable to distinguish between some of its visible events; this captures the notion that what a process can observe is an abstraction of the events in the underlying computation. Formally, a process is a pair (A,Π) where $A \subset E$ and Π is a partition of A. Only the events in A are visible to the process and the partition Π groups events in A into equivalence classes such that all events in an equivalence class are indistinguishable to the process. A process can only observe the equivalence class to which a visible event belongs, but not the event itself.

We note that an event may be visible to several processes; this denotes that the event is to take place simultaneously at all these processes. Furthermore, two events may be indistinguishable to a process p and distinguishable to another process q. We can model usual message passing systems by considering both active processes and channels as processes in our system. Similarly, shared variable systems may be modelled by considering active processes and shared variables as processes. It is important to note that we can define any pair (A,Π) to be a process. Our choice is dictated by what kinds of knowledge we wish to study. For instance, if we want to deduce the operation of a message passing system from the messages transmitted in the system, we will consider the channels (along which messages are transmitted) as processes.

Notation:: Unless otherwise stated, we use x,y,z to denote computations and p,q,r for processes; these symbols may be used with superscripts or subscripts also. The concatenation of sequences x and y will be denoted by $(x;y)$. For sequences x,y,$x \leq y$ denotes that x is a prefix of y; in this case (x,y) denotes the suffix of y obtained by deleting x from y. The empty sequence will be denoted by *null*.

The following example demonstrates how a process may be unable to distinguish some of its visible events.

Example

A mutual exclusion algorithm between two processes p,q is implemented by means of a token: only the process holding the token may enter a critical section. The decision by a process to enter its critical section is nondeterministic. If the token holder wishes to enter its critical section then it does so and sends the token to the other process upon completion of the critical section execution; if it decides not to enter the critical section, then it sends the token to the other process immediately. A portion of the system is shown in figure 1.

The set of events $E = \{a,b,c,d,e\}$. The relation $\rightarrow$ is $\{(a,c),(c,d),(a,d),(b,e)\}$ and the relation $\sim$ is $\{(a,b),(b,a)\}$. The important point to note is that the process receiving the token, i.e., with visible events d and e, cannot deduce whether the other process did enter its critical section prior to sending it the token, i.e., events d,e are indistinguishable to this process. Hence this process is defined by (A,Π) where $A = \{d,e\}$ and $\Pi = \{\{d,e\}\}$. The process sending the token is $(\{a,b,c\},\{\{a\},\{b\},\{c\}\})$. The process receiving the token views the receive as a single event, yet we model it as two distinct events d,e and represent the fact that the process cannot distinguish between them by having them in the same equivalence class.

Results

We simplify the discussion of distinguishability by assuming that each process has an associated set of colors, one distinct color for each equivalence class in its partition and distinct processes have no common color. Every event has all the colors of the various equivalence classes that it belongs to, corresponding to the processes to which it is visible. In any computation x, a process p cannot observe the invisible events and for each visible event it can only observe the color of the event. This is captured in the following definition.

Definition:: For a process p and a computation x, p's observations of x is the sequence of colors of the visible events of p in x.

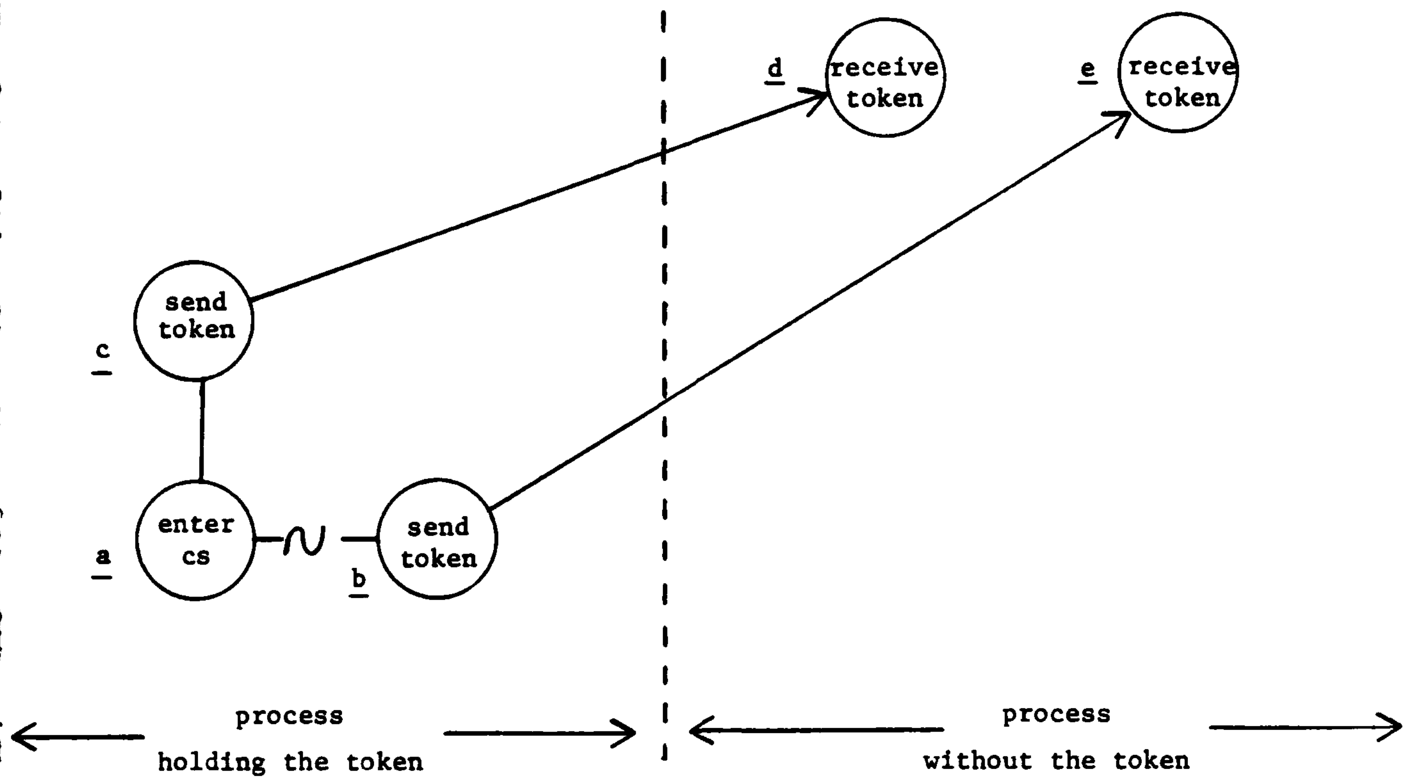


Figure 1:

All of our definitions and results are easily generalized when individual processes are replaced by sets of processes. This is because, processes $\{p,q\}$ together form a process whose visible events are the ones visible to either p or q and two visible events of this process are indistinguishable if and only if they are visible and indistinguishable to both p,q.

Definition:: A sequence x has a process chain $<p,q,...,r>$ if and only if there exists a subsequence of events, $e,e',...,e''$, not necessarily distinct, in x such that (1) $e \rightarrow e' \rightarrow ... \rightarrow e''$ and (2) e is on p, e' is $q,...,e''$ is on r.

A process chain as above indicates that in the given computation p informs q and later q informs $...,r$ is informed.

Definition:: For any process p, $[p]$ is an equivalence relation between computations defined as follows:

$$x\,[p]\,y \text{ means } p\text{'s observations of } x = p\text{'s observations of } y.$$ —

Intuitively, $x\,[p]\,y$, to be read as x is isomorphic to y with respect to p, means that p cannot distinguish between computations x,y. Isomorphism is the basis for our work; p's knowledge of the system has to be identical for both x,y.

Definition:: $x\,[p\,q\ldots r]\,y$ means there exists z such that,

$$x\,[p]\,z \text{ and } z\,[q\ldots r]\,y.$$

Intuitively, $x\,[p\,q]\,y$ denotes that there is a computation z which p cannot distinguish from x and q cannot distinguish from y. The relation $[p\,q]$ is the relational product of $[p]$ and $[q]$. This is generalized to arbitrary sequence of processes in $x[p\,q\ldots r]\,y$. A number of algebraic properties of isomorphism relations appear in [1]. Next, we present our fundamental theorem which relates process chains and isomorphism relations.

Theorem 1: (Fundamental Theorem) Let $x \le y$. For any sequence of processes $p,q,\ldots,r$, there is a process chain $<p\,q\ldots r>$ in (x,y) or $x\,[p\,q\ldots r]\,y$.

Process chains capture our intuitive notion of information transfer among processes. The theorem given above allows us to consider information transfer (or its absence) in algebraic terms using only isomorphism relations. In fact, our theorems about knowledge gain and loss are corollaries of this theorem. This theorem can be strengthened when every visible event of a process has a unique color, i.e., when a process can distinguish among all its visible event.

Theorem 2: Let $p,q,\ldots,r$ be processes which have unique colors for all their visible events, i.e., every equivalence class for each of these processes has exactly one event in it. Then, for any x,y where $x \le y$, $x[p\,q\,..\,r]\,y$ if and only if there is no process chain $<p\,q\,..\,r>$ in (x,y).

Now, we define knowledge predicates. Let b be any predicate whose value at a computation x is, *b at x*. We define a predicate *p knows b*.

Definition:: $(p \textit{ knows } b) \textit{ at } x =$ for all y: $x\,[p]\,y$: $b \textit{ at } y$.

Intuitively, *p knows b at x* if b is true for all computations which p cannot distinguish from x.

Note that b may itself be a knowledge predicate. The knowledge axioms appearing in [2] may be derived from this definition. It may be easily seen that $(p \textit{ knows } (q \textit{ knows } b)) \textit{ at } x =$ for all y: $x[p\,q]\,y : b \textit{ at } y$.

Notation:: We write,

p knows ... q knows b, to stand for, (*p knows ...* (*q knows b*))

Our next theorem shows that knowledge can be gained or lost only in a sequential manner.

Theorem 3: (*p knows ... q knows b at x* and $x\,[p \dots q]\,y$) implies *q knows b at y*.

Observe that x,y are arbitrary computations and p,...,q are arbitrary processes in the above theorem. If $x \leq y$, p knows ... q knows b at x and $\sim q$ knows b at y, then knowledge is lost and the theorem shows that $\sim x\,[p \dots q]\,y$. Using the fundamental theorem, we then deduce that there is a process chain $<p \dots q>$ in this case. Hence knowledge can be lost only by p informing the next process in the chain (of its intention to lose knowledge of b) which informs the next process, etc., until q is informed and q loses its knowledge of b. If $y \leq x$, $\sim q$ knows b at y and p knows ... q knows b at x, then knowledge is gained and the theorem then tells us that $\sim x\,[p \dots q]\,y$. Using algebraic properties of isomorphism relation this leads to $\sim y\,[q \dots p]\,y$ and then using the fundamental theorem, there is a process chain $<q \dots p>$ in (y,x). Therefore, knowledge is gained in a sequential manner in the reverse direction, q gaining knowledge of b and then informing the previous process in the chain of its knowledge of b,...,p gaining knowledge of b by being informed by the process ahead of it in the process chain.

This theorem gives a lower bound not only on the numbers of message transmissions but also on the lengths of the process chains. All the results in [1] may be similarly proven for the general model proposed in this paper. We sketch two new results.

K-Way Common Knowledge

The notion of common knowledge, as used here, is from Halpern and Moses [2]. They showed the impossibility of achieving common knowledge in a system which admits of no simultaneous events. We prove a stronger result: we show that if every event is visible to k or fewer processes, $k > 0$, then common knowledge among $k + 1$ processes cannot be gained or lost.

The predicate, *P has common knowledge of b*, where P is a set of processes and b is a predicate, has a value equal to the following expression at any computation x:

(b at x) **and** (for all p in P: *p knows b at x*) **and**

(for all p,q in P: *p knows q knows b at x*) **and** ...

It follows that, for any p in P,

P has common knowledge of b = p knows P has common knowledge of b.

Let every event e in a system be visible to at most k processes, i.e., no event occurs at more than k processes simultaneously. Then, we will show that for any P whose cardinality exceeds k and any predicate b, *P has common knowledge of b* is a *constant predicate*, i.e., it holds at all computations or its negation holds at all computations. In other words, no nontrivial common knowledge can be gained or lost. In particular, two-way communication is inadequate for achieving three-way synchronization *and gaining knowledge by each party about each other's knowledge about the synchronization.*

Theorem 4: Let every event in a system be visible to at most k processes and let P be a set of more than k processes. For every predicate b, *P has common knowledge of b*, is a constant predicate.

Proof (sketch): We show that,

P has common knowledge of b at x =

P has common knowledge of b at null.

Proof is by induction on length of x. Base step is trivial. For the inductive step, we need to show that for any computation $(x;e)$,

P has common knowledge of b at x = P has common knowledge of b at $(x;e)$.

From the premise of the theorem, there is some process p in P to which event e is invisible. Hence,

$x\,[p]\,(x;e)$.

Therefore, p knows b' at $x = p$ knows b' at $(x;e)$, for any b'. Letting b' be, *P has common knowledge of b*, and using the fact that,

P has common knowledge of b = p knows P has common knowledge of b,

the desired result follows.

The next result was suggested to us by Amir Pnueli. We show that all processes in an asynchronous message passing system can never agree that all channels are empty unless they are empty at all computations, i.e., they are initially empty and no process sends a message in any computation. A formal model of asynchronous message passing systems appears in [1] from which the following can be easily derived.

Let b be the predicate that all channels are empty. Then, $\sim q$ *knows* b *at* x *and* q *knows* b *at* $(x;e) \Rightarrow \sim b$ *at* x. Intuitively, if q gains knowledge of channel nonemptiness, it does so only by receiving a message (event e) and hence some channel was nonempty at x. Now we can prove:

Theorem 5: Let D denote the set of processes in an asynchronous message passing system in which some process sends a message in some computation. Let b be the predicate that all channels are empty. Then, for all p in D: p *knows* b, never holds.

Proof (sketch): If not, then there is a computation $(x;e)$ and process q such that $\sim q$ *knows* b *at* x and for all p in D: p *knows* b *at* $(x;e)$. Event e is a message receive on q. Therefore, there is a process r, $r \neq q$, such that

$$x\,[r]\,(x;e)$$

Hence, r *knows* b *at* x, because r *knows* b *at* $(x;e)$. From, $\sim q$ *knows* b *at* x and q *knows* b *at* $(x;e)$, we have $\sim b$ *at* x, which contradicts, r *knows* b *at* x.

In some sense, the simplest nontrivial knowledge that a process can acquire is whether an invisible event e has occurred. A process has this knowledge if and only if it knows that its observation includes a visible event e' where $e \rightarrow e'$. It follows that termination of one process p cannot be detected by another process q, because the event causing termination in p has no successor ($e \rightarrow e'$ means e' is a successor of e) in q.

When each event has a unique color, the question of whether an observation includes e', as above, is readily settled: we simply decide for each event in the observation whether it is a successor of e. However, introduction of colors makes this problem more difficult; we show that the problem is equivalent to answering whether an observation can occur in a system.

Theorem 6: Let x be a computation in system S and e be an invisible event of p.

p *knows* e has occurred *at* x = there exists y in S', $x\,[p]\,y$, where S' is a system derived from S by deleting all e', $e \rightarrow e'$.

—

Acknowledgement

We are immensely indebted to Professor Amir Pnueli for stimulating discussions about the "right" model for distributed computing. Professor C.A.R. Hoare's encouragement and advice about a more algebraic approach, is deeply appreciated.

References

1. Chandy, K. Mani and Misra, Jayadev, "How Processes Learn," *Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing*, Minaki, Canada, August 5-7, 1985 and *Distributed Computing*, Vol. 1, No. 1, 1985, (Springer-Verlag Publishing Company).

2. Halpern, Joseph Y. and Moses, Yoram, "Knowledge and Common Knowledge in a Distributed Environment," *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, Vancouver, Canada, August 27-29, 1984.

UNDERSTANDING A BYZANTINE ALGORITHM

J. Misra

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

TR-85-20 September 1985

This work was supported in part by a grant from the Office of Naval Research under grant ONR N00014-85-K-0057.

Understanding a Byzantine Algorithm

K. M. Chandy
J. Misra
University of Texas

Introduction

The problem of Byzantine Agreement defined in [1] is as follows. There are N processes any pair of which may communicate by messages. Any message sent is received instantly and correctly by the recipient. It is given that exactly t of the processes are faulty and the rest, N-t, are reliable. Each process is initially *off* or *on*. The problem is to devise a scheme whereby all reliable processes agree eventually on a common value, 0 or 1. Furthermore, the common value is 0(1) if all reliable processes are initially *off(on)*. Difficulty arises due to the nature of faulty processes: they may provide conflicting information in a concerted manner to thwart agreement by reliable processes.

We discuss an ingenious algorithm for this problem appearing in literature [2]. This note is intended as a different, and hopefully simpler, exposition of this algorithm and its proof. We believe that simplification is achieved by removing explicit message communication from the algorithm description. It should be easy to see how our scheme may be implemented using synchronous message communications. Our proof closely follows [2] though the restructuring results in some simplification.

It is known that solutions for this problem exist only if $N \geq 3t + 1$. We assume that $N = 3t + 1$ and $t > 0$. Let $low = t + 1$ and $high = 2t + 1$; therefore $high$ is the number of reliable processes. Observe that every subset of low processes has at least one reliable process and every subset of $high$ processes has at least low reliable processes.

Algorithm

We represent states of processes and their communication histories by a colored directed graph. Every vertex corresponds to a distinct process and a vertex state is *off/on* denoting the current state of the process. An edge (i,j) is directed from process i to j, $i \neq j$, and has a color, *black* or *white*.

Initially, there are no edges in the graph and a vertex state is the corresponding process state. The algorithm proceeds in rounds where during the first part of a round processes note the states of all other processes and edges that are present in the graph.

Upon completion of these observations, processes recompute their states and may add new outgoing edges. Note that states and outgoing edges of faulty processes may not be observed consistently by different reliable processes; this is the Byzantine aspect of the problem. We assume that some unspecified mechanism coordinates the observations and computations such that all observations precede all computations in a round. In particular, processes cannot observe changes in the graph or process states during the computation phase of a round.

Reliable processes use following rules to add edges, color edges and change their own states. These rules may be applied over and over by a process until no further rule is applicable. In the following, rules are given for a generic reliable process p and arbitrary process j; (p,j) is the edge from p to j, if it exists. Let *in(j)* denote the number of incoming edges to j, as observed by p, during a round; *white-out(p)* is the number of *white* outgoing edges of p.

Edge Addition::
(p,j) does not exist **and** (p observed j is *on* **or** $in(j) \geq low$) $\rightarrow$
add *black* edge (p,j)

Edge Coloring::
(p,j) is *black* **and** $in(j) \geq high$ $\rightarrow$ color (p,j) *white*

State Change::
{Let r be the round number}
p is *off* **and** *white-out(p)* $\geq t + r/2$ $\rightarrow$ p becomes *on*

Observation

1. No reliable process becomes *off* once it is *on*.

2. No edge is ever deleted by a reliable process. No *white* outgoing edge of a reliable process ever becomes *black*.

3. A reliable process creates a *black* edge (p,j) only if j is observed *on* by some reliable process, possibly p. edge (p,j) is *white* only if there are at least *low* reliable processes with edges to j and hence these edges are observed in all subsequent rounds by all reliable processes.

4. There is nothing magical about the function $t + r/2$.

Let R be such that agreement on a common value is reached by the end of round R. Any function f satisfying the following, is acceptable.

$f(2) = low$, $f(r+2) \leq 1 + f(r)$ for all r in $0 \leq r \leq R-2$, $f(R-3) \geq$ *high*, $f(R-2) \geq$ *high*.

We allow function f to be real valued and hence, without loss in generality, we can relax one of the conditions to: $\lceil f(R-3) \rceil \geq$ *high*. One of our goals is to minimize R. Note that $\lceil f \rceil$ increases from *low* to at least *high* from round 2 to $R-3$ and f can increase by at most 1 in two rounds. The unique minimum for R is $2t+4$ and a choice for f is, $t + r/2$.

We have not yet specified the conditions under which processes commit to different values. These conditions become apparent from the results proven below. In the following p,q denote reliable processes and j an arbitrary process. We use "at round r" to mean upon completions of computations of round r and "in round r" to mean prior to computations of that round. "At round 0" will refer to initial conditions.

Lemma 1:

Edge (p,q) is *white* at round $(r+2)$ iff q is *on* at round r.

Proof:

If q is *on* at round r, it is observed *on* in round $(R+1)$ by all reliable processes and hence $in(q) \geq$ *high* at $(r+1)$. Then every reliable process, including p, has a *white* edge to q at round $(r+2)$. Conversely, if q is *off* at round r, it is *off* at all precious rounds and it is observed *off* in round $(r+1)$. Hence $in(q) <$ *low* at $(r+1)$ and therefore no reliable process has a *white* edge to q at $(r+2)$.

Let $np(r)$ denote the number of reliable processes which are *on* at round r. We show in the following lemma that if any reliable process changes state then every reliable process is *on* two rounds later; furthermore a state change is possible only if at least $r/2$ reliable processes are *on* two rounds earlier. We note that *np(r)* is monotone nondecreasing in r.

Lemma 2:

For every r, $2 \leq r \leq R-2$,

$[np(0) = np(r)]$ or $[np(r+2) = high$ and $np(r-2) \geq (r-1)/2]$

Proof:

Consider the smallest r, if any, for which $np(0) \neq np(r)$. If no such r exists, the lemma holds. Some reliable process p applies the state change rule at round r. For p, *white—out*$(p) \geq t + r/2$ in round r; hence p has *white* edges to at least $r/2$ reliable processes and, from lemma 1, all these are *on* at round $(r-2)$, i.e. $np(r-2) \geq r/2$. Also $np(r) > np(r-2)$ and hence $np(r) > r/2$.

Next we show that every reliable process is *on* at round $(r+2)$. We only need to prove this for reliable processes which are *off* at round r; let q be one such process. We show that if (p,j) is *white* at round r then (q,j) is *white* at round $(r+2)$: for (p,j) to be *white*, $in(j) \geq high$ in some round before or in round r, as observed by p; hence at least *low* reliable processes have edges to j in round r; then every reliable process has at least a *black* edge to j at $(r+1)$ and *white* edge at $(r+2)$. Also, edge (p,q) is *white* at round $(r+2)$, from lemma 1. Also, from lemma 1, p has no *white* edge to q at round r. Therefore, *white-out(q)* at round $(r+2) \geq 1 +$ *white-out(p)* at round $r \geq t + (r+2)/2$; hence q is *on* at $(r+2)$.

Consider any round r'. For $r' < r$, $np(r') = np(0)$ and hence the lemma holds.

For $r' \geq r$, $np(r'+2) = high$.
For $r' = r+1$, $np(r'-2) \geq np(r-2) \geq r/2 = (r'-1)/2$.
For $r' = r+2$, or $r' = r+3$, $np(r'-2) \geq np(r) \geq np(r-2)+1 \geq (r+2)/2 \geq (r'-1)/2$.
For all larger values r', $np(r'-2) = high \geq (r-1)/2$.

Theorem:

1. $np(0) = 0$ **implies** $np(R-2) = O$
2. $np(0) \geq low$ **implies** $np(R-2) = high$
3. $np(0) < low$ **implies** $[np(0) = np(R-2)$ **or** $np(R-2) = high]$

Proof:

1. Suppose $np(0) = 0$. Observe that $np(1) = 0$. Let r be the smallest value, $r \geq 2$, for which $np(0) \neq np(r)$. Then from lemma 2, $np(r-2) \geq (r-1)/2 > 0$, contradiction.
2. Let $np(0) \geq low$. From lemma 1, every reliable process p is *on* at round 2 because *white-out(p)* $\geq low = t + r/2$, at $r = 2$. Hence $np(2) = high$, from which the result follows.
3. Suppose $np(0) \neq np(R-2)$. Then from lemma 2, $np(R-4) \geq (R-3)/2 \geq (2t+1)/2$. Hence $np(R-4) \geq low$. Because $np(0) < low$, $np(0) \neq np(R-4)$. From lemma 2, $np(R-2) = high$.

Commit Rule:: At round R,
white−out$(p) \geq high \rightarrow$ commit to 1

white–out$(p) < high \rightarrow$ commit to 0

Corollary 1:

All reliable processes commit to the same value. If they are initially *off/on* they commit to 0(1).

Proof:

From the theorem $np(R-2) < low$ or $np(R-2) = high$. If $np(R-2) < low$ then, from lemma 1, for any reliable process p, *white–out*$(p) < high$ at round R. If $np(R-2) = high$ then, again from lemma 1, *white–out*$(p) \geq high$. Hence all reliable processes commit to the same value. Other parts follow trivially from the theorem.

REFERENCES

1. Lamport, L., Shostak, R. and Pease, M. "Byzantine Generals Problem", TOPLAS 1982.

2. Lynch, N., Fischer, J. and Fowler, R. "A Simple and Efficient Byzantine Generals Algorithm", Proceedings of the 2nd Symposium on Reliability in Distributed Software and Database Systems, July 1982.

On the Nonexistence of Robust Commit Protocol

K. Mani Chandy
Jayadev Misra

Department of Computer Sciences
The University of Texas
Austin, Texas 78712
(512)471-4353

21 November 1985

This research was supported by a grant from the Office of Naval Research under grant N00014-85-K-0057.

Table of Contents

1. Introduction

An important result in the theory of asynchronous message passing systems is the impossibility of distributed consensus with one faulty process. This problem was first defined and its impossibility proven in [2]. The proof in [2] relied on operational details of asynchronous message communication: channels, sends and receives, etc. Our goal is to prove this beautiful result in, what Dijkstra terms, a purely nonoperational framework. We do so by defining a set of properties asynchronous message passing systems and a set of requirements for consensus with, or without , faulty processes. Our formulation is slightly more general than the original formulation: we allow processes to be nondeterministic and we do not require any kind of fairness after failure of a process.

We use some algebraic properties of system computations introduced in [1]; we also use most of the key ideas from [2].

2. Asynchronous Message Passing Systems

Discussions of asynchronous message passing systems are usually given in terms of processes, channels, send and receive primitives for message communications, etc. We take a different approach; we define these systems by a small set of their properties which make no mention of channels or messages.

An asynchronous message passing system, to be henceforth called *system*, is a set of *processes* and a set of *computations*. A computation is a finite sequence of pairs of the form (e,p), where e is an *event* and p is a process.

An intuitive meaning of a computation in a system is that it is possible for every event to happen at the corresponding process in the sequence given by the computation. We assign no meanings to events or processes. No causality among events, such as between sends and receives, is explicitly stated. It is not required that the processes be deterministic.

Property A1:: Every prefix of a computation is a computation.

The empty sequence is, therefore, a computation; it is denoted by *null*.

Notations:: Symbols x, y, z, y' denote computations, e, e' events and p, a process. $<x;(e,p)>$ denotes the sequence obtained by concatenating the pair (e,p) to x. An *extension* of a computation x is a computation of which x is a prefix. For any x and p let x_p be the subsequence of x containing p as the process component. There is an event on p between x, y, where y is an extension of x, if there is some pair (e,p) in y after x.

Definition:: Computations x, y are *isomorphic* with respect to p, to ~~be~~ denoted by $x\,[p]\,y$, means that $x_p = y_p$.

The notion of isomorphism is from [1] where it was used to state and derive several properties of system computations. For this paper we only note the following two elementary properties.

- $[p]$ is an equivalence relation over system computations.
- For x a prefix of y, there is an event on p between x, y iff $\neg\, x\, [p]\, y$.

Property A2:: Let $<x; (e,p)>$ be a computation and y an extension of x such that $x\, [p]\, y$. Then, $<y; (e,p)>$ is a computation.

The intuitive meaning of property A2 is that if an event e can happen at a process p at some point in the computation of the system then the same event can happen at a later point in the computation, provided that p has taken no other step between these two points. This requirement does not hold for all concurrent systems; in a shared variable system, a process reading the value of a shared variable is not guaranteed to read the same value at a later point. However communications in message passing systems are limited to message sends and receives which, by their asynchronous nature, satisfy this property.

Property A3:: For any x and p, there is a computation $<x; (e,p)>$.

In a message passing system, processes wait only to receive messages; this property postulates that a process can always terminate its waiting without receiving a message. Furthermore, a terminated process can always take a dummy step.

We have defined computations as finite sequences and almost all of our proofs will exploit the finiteness assumption. In order to state the problem formally, however, we need the concept of a *fair sequence*, a special kind of infinite sequence. A fair sequence is an infinite sequence of (event, process) pairs where each finite prefix is a computation and each process appears in an infinite number of pairs. It follows, by repeated application of A3, that every computation is a prefix of some fair sequence.

3. Commit Protocol

Intuitive Discussion

A commit protocol is a system in which every process eventually (explained below) commits to a value, 0 or 1, and all processes commit to the same value. Furthermore, processes do not commit to one value, say 0, in all computations. Fair sequences capture our intuitive notion of infinitely long computations (though, recall that our formal model only admits of finite computations) and hence, we require that every fair sequence have a finite prefix in which a commitment is made. Since every computation is a prefix of some fair sequence, it follows that every computation has an extension which commits.

Now we introduce the extremely useful idea of *valency* of a computation, from [2]. Call a computation 0—*valent* if all extensions of it which commit , commit to 0; similarly 1-valent. A *bivalent* computation is neither 0-valent nor 1-valent . Since every computation has an extension which commits, it follows that a bivalent computation has a 0-valent extension and a 1-valent extension. Call a computation *univalent* if it is either 0-valent or 1-valent.

Now we give a formal set of requirements for a commit protocol.

Formal Description of Commit Protocol

Let *y includes x* denote that x_p is a prefix of y_p for all p, i.e., every process has extended its own computation in going from x to y. The relation *includes* is a generalization of extension.

A commit protocol is a system in which computations are 0-valent, 1-valent or bivalent. These satisfy:

C1:: There is a 0-valent computation and a 1-valent computation.

C2:: Every bivalent computation has an *extension* that is 0-valent and an extension that is 1-valent.

C3:: If y includes x and x is 0-valent (1-valent) then y is 0-valent (1-valent).

C4:: Every fair sequence has a finite prefix that is univalent.

Observation:: The null computation is bivalent, from C1 and C3.

Definition:: Two computations are *incompatible* means that one is 0-valent and the other is 1-valent; they are *compatible* otherwise.

Lemma 1:: For a computation $<x;(e,p)>$ and any extension z of x,

$$\neg\, x\,[p]\,z \text{ or the computations } <x;(e,p)>,\ z \text{ are compatible.}$$

Proof:: Suppose $x\,[p]\,z$. Then, from A2, $<x;(e,p)>$ is a computation. If $<x;(e,p)>$ and z are incompatible then one is 0-valent and the other 1-valent and hence $<z;(e,p)>$, which includes both these computations, is both 0-valent and 1-valent, from C3; contradiction! □

A process p is a decider for a bivalent computation x if an event on p extends x to a 0-valent computation and another event on p extends x to a 1-valent computation; equivalently, there exist incompatible computations $<x; (e,p)>$ and $<x; (e',p)>$. The theorem, given below, shows that every commit protocol has such a pair of incompatible computations.

Theorem 1:: (Existence of Decider)

There exist incompatible computations $<x; (e,p)>$ and $<x; (e',p)>$.

Proof:: We assume the contrary: every pair of computations $<x; (e,p)>$, $<x; (e',p)>$ is compatible. We then show that for every bivalent x and p there exists a bivalent extension y of x such that $\neg\, x\, [p]\, y$; we then show that this leads to a contradiction of the requirement C4.

Consider any bivalent x and process p. There is a computation $<x; (e,p)>$, from A3. If $<x; (e,p)>$ is bivalent, the result is proven. Otherwise, without loss in generality, assume that $<x; (e,p)>$ is 0-valent. From bivalence of x, and C2, there exists a 1-valent extension z of x. Since $<x; (e,p)>$ is 0-valent and z is 1-valent, they are incompatible and hence, from lemma 1, $\neg\, x\, [p]\, z$.

Now we display a bivalent extension y of x for which $\neg\, x\, [p]\, y$. We only consider extensions of x which are also prefixes of z. Since $x\, [p]\, x$ and $\neg\, x\, [p]\, z$, there exist extensions y', y such that $x\, [p]\, y'$, $\neg\, x\, [p]\, y$ and y is a one event extension of y'. Therefore, $y = <y'; (e', p)>$ for some e'. We show that y is neither 0-valent nor 1-valent and hence bivalent. From C2, y is not 0-valent because z includes y and z is 1-valent. To see that y is not 1-valent, note: (1) since $<x; (e,p)>$ is a computation, y' is an extension of x and $x\, [p]\, y'$, from A2, $<y'; (e,p)>$ is a computation, (2) $<y', (e,p)>$ and $y = <y', (e', p)>$ are compatible, from the assumption at the beginning of this proof, and, (3) $<y', (e,p)>$ is 0-valent, from C3, because it includes $<x, (e,p)>$ and the latter is 0-valent and, (4) from (2) and (3), y is not 1-valent.

Now we have a procedure for obtaining a longer bivalent computation from any bivalent computation and any process p. We may apply this procedure infinitely often, starting from null computation which is bivalent, using an arbitrary process p each time and ensuring that every process is chosen an infinite number of times. The resulting infinite sequence is fair and all its finite prefixes are bivalent. This contradicts requirement C4. □

4. Robust Commit Protocol

A commit protocol is *robust* means that in spite of failure of any one process at any point in the computation the remaining processes can commit to a value. Failure of a process can be modelled by no event happening at that process. It is somewhat more difficult to define a fair sequence after failure of a process; fortunately we don't need to define that concept. We can work with a very weak requirement for robustness: for any p, it should be possible to extend a bivalent computation to a univalent computation without any event happening on p in between. Formally,

R:: For every bivalent x and p, there exists a univalent extension z of x such that $x\,[p]\,z$.

We now show that no robust commit protocol has a decider for any bivalent computation.

Lemma 2::

In any robust commit protocol, any two computations $y = <x;(e,p)>$ and $y' = <x;(e',p)>$ are compatible.

Proof::

If x is univalent then y, y' are compatible, using C3. For any bivalent x and p, apply (R) to conclude that there is a univalent extension z of x such that $x\,[p]\,z$. From lemma 1, y,z are compatible and also y',z are compatible. Since z is univalent, y,y' are compatible.

Theorem 2:: **(Impossibility of Robust Commit)**

There is no robust commit protocol.

Proof::

Immediate from theorem 1 and lemma 2. □

Acknowledgement

We are greatly indebted to members of the Austin Tuesday Afternoon Club for a thorough reading of an earlier draft of this manuscript.

References

1. Chandy, K. Mani and Misra, Jayadev, "How Processes Learn," *Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing*, Minaki, Canada, August 5-7, 1985 and to appear in *Distributed Computing*, in 1985.

2. Fischer, Michael J., Lynch, Nancy A., and Paterson, Michael S., "Impossibility of Distributed Consensus with One Faulty Process," MIT/LCS/TR-282, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA.

acm

Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing

[illegible] Ontario, Canada

[illegible] 7, 198

[illegible] THE ACM
[illegible] INTEREST GROUP FOR AUTOMATA AND COMPUTABILITY THEORY AND
[illegible] INTEREST GROUP FOR OPERATING SYSTEM PRINCIPLES

HOW PROCESSES LEARN

K. Mani Chandy & Jayadev Misra
Department of Computer Sciences University of Texas Austin, 78712

1. Introduction

Processes in distributed systems communicate with one another exclusively by sending and receiving messages. A process has access to its state but not to the states of other processes. Many distributed algorithms require that a process determine facts about the overall system computation. In anthropomorphic terms, processes "learn" about states of other process in the evolution of system computation. This paper is concerned with how processes learn. We give a precise characterization of the minimum information flow necessary for a process to determine specific facts about the system.

The central concept in our study is that of *isomorphism* between system computations with respect to a process: two system computations are isomorphic with respect to a process if the process behavior is identical in both. In anthropomorphic terms, "system computations are isomorphic with respect to a process" means the process cannot distinguish between them.

Many correctness arguments about distributed systems have the following operational flavor: "I will send a message to you and then you will think that I am busy and so you will broadcast ... ". Such operational arguments are difficult to understand and error prone. The basis for such operational arguments is usually a "process chain": a sequence of message transfers along a chain of processes. We advocate nonoperational reasoning. The basis for nonoperational arguments is isomorphism; we relate isomorphism to process chains. Algebraic properties of system computations under isomorphism provide a precise framework for correctness arguments.

It has been proposed [3,6] that a notion of "knowledge" is useful in studying distributed computations. In earlier works, knowledge is introduced via a set of axioms [4]. Our definition of knowledge is based on isomorphism. Our model allows us to study how knowledge is "gained" or "lost". One of our key theorems states that knowledge gain and knowledge loss both require sequential transfer of information: if process q does not know fact b and later, p knows that q knows b, then q must have communicated with p, perhaps indirectly through other processes, between these two points in the computation; conversely, if p knows that q knows b and later, q does not know b then p must have communicated with q between these two points in the computation. In the first case, the effect of communication is to inform p of q's knowledge of b. Analogously, in the second case, the effect of communication is to inform q of p's intention of relinquishing its knowledge (that q knows b). Generalizations of these results for arbitrary sequences of processes are stated and proved as corollaries of a general theorem on isomorphism.

We use the results alluded to in the last paragraph for proving lower bounds on the number of messages required to solve certain problems. We show, for instance, that there is no algorithm to detect termination of an underlying computation using only a bounded number of overhead messages.

2. Model of a Distributed System

A distributed system consists of a finite set of processes. A process is characterized by a set of process computations each of which is a finite sequence of events on that process. Process computations are prefix closed, i.e. all prefixes of a process computation are also process computations (of that process). An event on a process is either a *send*, a *receive* or an *internal event*. A *send* event on a process corresponds to sending of a message to another process. A *receive* event on a process corresponds to reception of a message by the process. There is no external communication associated with an *internal* event. For a set of processes P, a send event by P is a send event by some component process of P to a process outside P; similarly a receive event by P denotes receipt by some process in P of a message sent from outside P. Communication among processes in P are internal events of P. We use "e is on P", for event e and process set P, to denote that e is an event on some process in P. We rule out processes which have no event in any computation. We assume that all events and all messages are distinguished; for instance, multiple occurrences of the same message are distinguished by affixing sequence numbers to them.

Let z be any sequence of events on component processes of a distributed system. The *projection of z* on a component process p, denoted by z_p, is the subsequence of z consisting of all events on p. A finite sequence of events z is a *system computation* of a distributed system means (1) for all processes p, z_p is a process computation of p and, (2) for every receive event in z, say receipt of message m by process p, there is a send event, of sending m to p, which occurs earlier than the receive in z: this send event will be called the send event *corresponding* to the receive. We leave it to the reader to show that system computations are prefix closed.

In this paper we consider a single (generic) distributed system. For instance, when we say "z is a computation" we mean that z is a computation of the distributed system considered here. We use *computation* to mean *system computation* when no confusion can arise.

Notation: We use x, y, z to denote computations, p, q for processes and P, Q for process sets; these symbols may be used with subscripts or superscripts. The concatenation of two sequences y and z will be denoted by $(y;z)$. For sequences y and z, $y \leq z$ denotes that y is a prefix of z; in this case (y, z) denotes the suffix of z obtained by removing y from z. The empty sequence will be denoted by *null*. The symbol $=$ is used to denote equalities among sets and among predicates. The symbol $\equiv$ is used for definitions. The set of all processes in the system will be denoted by D and for any process set P, $\overline{P} = D - P$.

3. Isomorphism

We define the relation $[p]$ on the set of system computations as follows.

Definition: For system computations x,y:
$x\,[p]\,y \equiv (x_p = y_p)$.

In other words, $x\,[p]\,y$ means p's computation is the same in system computations x and y. In this case, we say x, y are *isomorphic with respect to p*. For a process set P, define relation $[P]$, on the system computations, as follows.

Definition: $x\,[P]\,y \equiv$ for all p in P, $x\,[p]\,y$.

Thus $x\,[P]\,y$ means that, given only the computations of processes in P we cannot distinguish x from y. From definition, $x\,[\{\ \}]\,y$, for all computations x, y where $\{\ \}$ denotes the empty set. Observe that $[P]$ is an equivalence relation.

It is convenient to represent all such isomorphism relations by an *isomorphism diagram*: an undirected labelled graph whose vertices are computations and there is an edge labelled $[P]$ between vertices x, y if P is the largest set of processes for which $x\,[P]\,y$. Observe that every vertex has a self loop labelled $[D]$ where D is the set of all processes in the system. Note that $x\,[D]\,y$, $x \neq y$, implies y is a permutation of x.

Example 1: Consider a system with two processes, p *and* q, for which part of the isomorphism diagram, showing the relationships among four system computation, is given below.

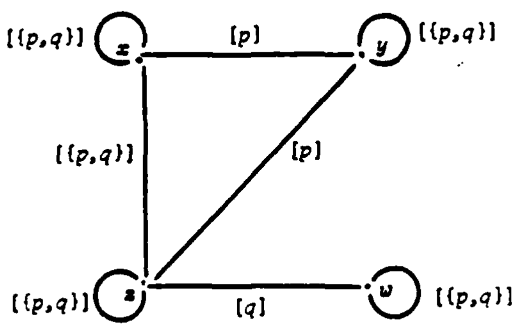


Figure 3-1: An Isomorphism Diagram

From the diagram $x\,[p]\,y$, but not $x\,[q]\,y$. This means p has the same computations in both x and y, whereas q's computations in x and y differ. Computations x and z have the same computations for both p and q; hence one is a permutation of the other. There is no direct relationship between y and w; neither $y\,[p]\,w$ nor $y\,[q]\,w$ holds. However, there is an indirect relationship between y and w because $y\,[p]\,z$ and $z\,[q]\,w$. We explore such indirect relationships next.

□

Definition: Let $n > 0$ and P_i be process sets, $0 \leq i \leq n$.

$x\,[P_0 \ldots P_n]\,z \equiv x\,[P_0 \ldots P_{n-1}]\,y$ *and* $y\,[P_n]\,z$, for some computation y.

Hence, $[P\,Q] = [P] \circ [Q]$ where "$\circ$" is the relational composition operator. This operator is associative (from properties of relations). In terms of the isomorphism diagram, $x[P_0 \ldots P_n]\,z$ means there is a path from x to z whose edges are labelled with $Q_0, \ldots, Q_n$, respectively, where $Q_i \supseteq P_i$, for all i.

Example 1 (contd.): We have $y\,[p\ q]\,w$ and $w\,[q\ p]\,y$. Also, trivially, $y\,[q\ p]\,z$, $y\,[q\ p\ q]\,z$, etc.

□

We note some properties of isomorphism relations. In the following, $P, P_1, \ldots, P_n, Q$, denote arbitrary process sets and x, y, z denote arbitrary computations.

1. $[P]$ is an equivalence relation.
2. (Substitution) $([\beta] = [\delta])$ *implies* $([\alpha\ \beta\ \gamma] = [\alpha\ \delta\ \gamma])$ for arbitrary sequences of process sets $\alpha, \beta, \gamma, \delta$.
3. (Idempotence) $[P\,P] = [P]$
4. (Reflexivity) $x\,[P_1 \ldots P_n]\,x$
5. (Inversion) $x\,[P_1 \ldots P_n]\,y = y\,[P_n \ldots P_1]\,x$
6. (Concatenation) For $0 < m < n$,

   $\exists y$: $x\,[P_1 \ldots P_m]\,y$, $y\,[P_{m+1} \ldots P_n]\,z = x\,[P_1 \ldots P_m\,P_{m+1} \ldots P_n]\,z$
7. $[P \cup Q] = ([P] \cap [Q])$
8. $(Q \supseteq P) = ([Q] \subseteq [P])$
9. $(P = Q) = ([P] = [Q])$
10. $Q \supseteq P$ *implies* $([Q\,P] = [P] = [P\,Q])$

These properties follow from properties of relations and our model. We only sketch a proof of one part of property 8:

$([Q] \subseteq [P])$ *implies* $(Q \supseteq P)$.

If $Q \not\supseteq P$ then there is a process p in $P - Q$. From our model, p has an event e in some computation $(x;e)$. Then $x\,[Q]\,(x;e)$ and $\sim x\,[P]\,(x;e)$. Hence $[Q] \not\subseteq [P]$.

3.1. Process Chains

As noted in the introduction, the basis for many operational arguments are process chains: process p informing q which in turn, informs r etc. One of our goals is to replace such concepts by algebraic properties of system computations. In this section, we show how process chains are related to isomorphism. We first define process chains; this definition is along the lines suggested by Lamport [5].

Definition: For events e, e' in a computation z, $e \xrightarrow{z} e'$ means:

1. e' is a receive and e is the corresponding send, or
2. events e, e' are in the same process computation and ($e = e'$ or e occurs earlier than e'), or
3. there exists an event e'' such that $e \xrightarrow{z} e''$ and $e'' \xrightarrow{z} e'$.

For brevity we write $e \rightarrow e'$ when the computation z is understood from context. We will write $e_0 \rightarrow e_1 \rightarrow \ldots e_{n-1} \rightarrow e_n$, as shorthand for $e_0 \rightarrow e_1$ and ... and $e_{n-1} \rightarrow e_n$. Observe that $e \rightarrow e$ for every event e in z. A computation z has a *process chain* $<P_0\ P_1\ \ldots\ P_n>$ means there exist events $e_0, e_1, \ldots e_n$, not necessarily distinct, in z such that event e_i is on P_i, for all $0 \leq i \leq n$, and $e_0 \rightarrow e_1 \rightarrow \ldots \rightarrow e_n$.

Observation 1: Any occurrence of "P " in a process chain may be replaced by "$P\ P$", or vice versa, since for any event e on P, $e \rightarrow e$.

Observation 2: Let x be a sequence consisting of a subset of events from a computation y. Suppose that for every event e in x, every e', where $e' \xrightarrow{y} e$, is also in x and $e' \xrightarrow{x} e$. Then x is a computation.

3.2. Relationship Between Isomorphism and Process Chain

Theorem 1: (Fundamental Theorem of Process Chains)

Let z be a computation and x a prefix of z. Let P_1, P_2 ... P_n, $n \geq 1$, be sets of processes. Then $x\,[P_1\,P_2 \ldots P_n]\,z$ or there is a process chain $<P_1\,P_2 \ldots P_n>$ in (x,z).

□

Proof: Omitted

□

3.3. An Application of Isomorphism: How To Construct A Computation By Fusing Separate Ones

In this section, we show an application of isomorphism: we give a construction to "fuse" two computations to obtain a new computation, provided certain types of paths exist in the isomorphism diagram. We motivate the discussion by the following observations. Suppose $(x;E)$ and $(x;\overline{E})$ are computations where all events in E are on a process set P and all events in $\overline{E}$ are on $\overline{P}$. Then, from definition, $(x;\overline{E};E)$ and $(x;E;\overline{E})$ are also computations, because events in $E,\overline{E}$ are independent and hence may be fused in arbitrary order. A similar result appears in Fischer, Lynch and Paterson [2]. The following lemma is a generalization of this observation.

Lemma 1: Let x, y, z be computations where $x \leq y$ and $x \leq z$. Let P, Q be such that $P \cup Q = D$, $x\,[P]\,y$ and $x\,[Q]\,z$. Then there exists a computation w where $x \leq w$, $y\,[Q]\,w$ and $z\,[P]\,w$.

□

The relationships among x, y, z and w are represented by the following commutative isomorphism diagram.

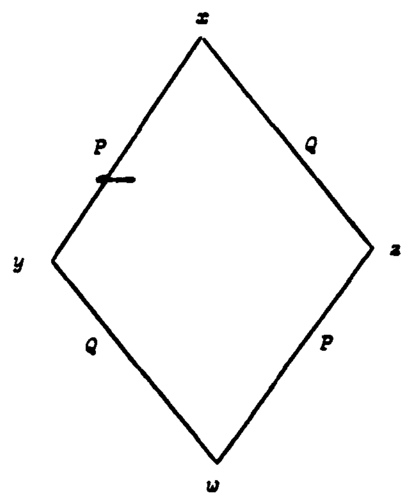


Figure 3-2: Isomorphism Diagram Depicting Fusion

Proof of the Lemma:
Let $w = x; (x,y); (x,z)$.

From the condition of the lemma, (x, y) has events only on $\overline{P}$ and (x, z) has events only on $\overline{Q}$. Since $P \cup Q = D$, $\overline{P} \cap \overline{Q} = \{ \}$ and hence no process has events in both (x,y) and (x,z). It follows, from definition of computations, that w is a computation. Also $y\,[\,Q\,]\,w$, $z\,[\,P\,]\,w$ and $x \leq w$, as required for proof of the lemma.

□

Note that, in the construction of lemma 2, all events from E and $\overline{E}$ were present in the fused computation. We prove a far more general result below. We show that for any two arbitrary computations y and z, the projected computations, y_P and $z_{\overline{P}}$, may be fused to form a new computation provided there is a computation x which is a prefix of both y and z, and no message sent by $\overline{P}$ in (x,y) is received by P in (x,y) and no message sent by P in (x,z) is received by $\overline{P}$ in (x,z). This makes intuitive sense: processes in P can execute all events in y given only that processes in $\overline{P}$ execute all events up to x and similarly for executions of events on $\overline{P}$ up to z. However, the statement and proof of this result are difficult without the notion of isomorphism. We note that the result may be easily generalized to fusions of arbitrary numbers of computations under similar constraints.

Theorem 2: (Fusion of Computations): Consider system computations x, y, z where $x \leq y$ and $x \leq z$. Let P be a set of processes such that there is no process chain, (1) $<P\,\overline{P}>$ in (x, y) and (2) $<\overline{P}\,P>$ in (x, z). Then there is a computation w where, $x \leq w$, $y\,[\,\overline{P}\,]\,w$ and $z\,[\,P\,]\,w$. That is, w consists of all events on $\overline{P}$ from y and all events on P from z.

□

Proof of the Theorem: According to theorem 1, absence of process chains as given in this theorem means that, $x\,[\,P\,\overline{P}\,]\,y$ and $x\,[\,\overline{P}\,P\,]\,z$.

Consider the isomorphism diagram in Fig. 3-3. Label the intermediate point between x, y as u and between x, z as v in this figure. Now we apply lemma 1 to x, u, v to obtain w. Note that, $u\,[\,\overline{P}\,]\,y$ and $u\,[\,\overline{P}\,]\,w$; hence $y\,[\,\overline{P}\,]\,w$. Similarly $z\,[\,P\,]\,w$. This proves the theorem.

□

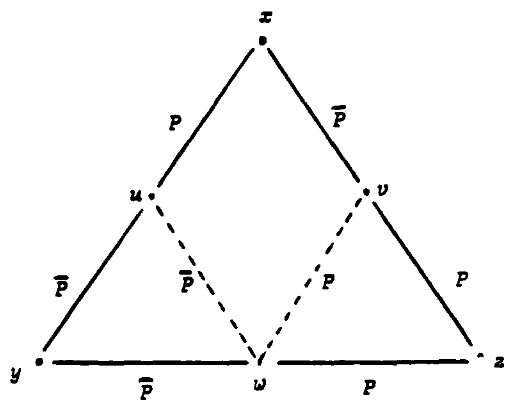


Figure 3-3: Isomorphism Diagram Depicting Proof of Fusion Theorem

3.4. Semantics Of Event Types In Terms Of Isomorphism

We now use isomorphism to state and derive some important facts about various types of events. First, note that a process carries out an internal event or sends a message depending on its own computation alone. Therefore, if a process takes such a step in a computation x, it will also do so in y, if x, y are isomorphic with respect to this process. An analogous result holds for internal and receive events. The following principle, which states these facts formally, may be proven from the definition of system computation.

Principle of Computation Extension::

Let e be an event on P.

1. e is an internal or send event: ($x\,[P]\,y$ *and* $(x;e)$ is a computation) *implies* $(y;e)$ is a computation

2. e is an internal or receive event: $(x;e)\,[P]\,y$ *implies* $(y-e)$ is a computation, where $(y-e)$ is the sequence obtained by deleting e from y.

□

Note: In (1), $(x;e)\,[P]\,(y;e)$ and in (2), $x\,[P]\,(y-e)$.

Corollary: Let e be a receive event on P and let the corresponding send event be on Q.

($x\,[P \cup Q]\,y$ *and* $(x;e)$ is a computation) *implies* $(y;e)$ is a computation.

□

Proof: e is an internal event of $P \cup Q$.

□

Following theorem follows from the principle of computation extension.

Theorem 3: Let $(x;e)$ be a computation where e is an event on P.

Case 1: e is a receive:

for every z: $(x;e)\,[P\overline{P}]\,z$ *implies* $x\,[P\overline{P}]\,z$

Case 2: e is a send:

for every z: $x\,[P\overline{P}]\,z$ *implies* $(x;e)\,[P\overline{P}]\,z$

Case 3: e is an internal event:

for every z: $(x;e)\,[P\overline{P}]\,z = x\,[P\overline{P}]\,z$

□

Proof: We will prove only Case 2; other cases are similarly proven.

$x\,[P\overline{P}]\,z$ *implies* there exists y, $x\,[P]\,y$ *and* $y\,[\overline{P}]\,z$. From principle of computation extension, $(y;e)$ is a computation and $(x;e)\,[P]\,(y;e)$.

Also, $(y;e)\,[\overline{P}]\,y$.

Hence, $(x;e)\,[P\overline{P}\,\overline{P}]\,z$ and therfore, $(x;e)\,[P\overline{P}]\,z$.

□

This theorem captures the intuitive notion that the set of possible computations, isomorphic with respect to P, can only shrink in size as a result of a reception as computations which do not include the corresponding send are ruled out. Similarly, the set of possible computations, isomorphic with respect to P cannot shrink as a result of a send: after the send, additional computations which accept the message sent are isomorphic while all prior isomorphic computations remain isomorphic. An internal event can neither expand nor shrink the set of isomorphic computations.

4. Knowledge

As we have remarked earlier, predicates of the type P *knows* b *at* x may be defined using isomorphism. We explore properties of such predicates in our model. We show that they satisfy the "knowledge axioms" as given in [3,6]. We prove a general result which shows that certain forms of knowledge can only be gained or lost in a sequential fashion along a chain of processes. That is, if b is false for a computation and later, P_1 *knows* P_2 *knows* $\ldots$ P_n *knows* b (this represents knowledge gain), then there is a process chain $<P_n\, P_{n-1} \ldots P_1>$ between these two points of the computation. Conversely, if P_1 *knows* P_2 *knows* $\ldots$ P_n *knows* b and later, b is false (this represents knowledge loss), then there is a process chain $<P_1\, P_2 \ldots P_n>$ between these two points of the computation.

Crucial to our work is the notion of *local predicates*: a predicate local to p can change in value only as a result of events on p. We show that local predicates play a key role in understanding knowledge predicates.

4.1. Knowledge Predicates

Let b denote a predicate on system computations and "b *at* x" its value for computation x. Our predicates are *total*, i.e. for each x, b *at* x is either *true* or *false*. We furthermore assume that $x\,[D]\,y$ *implies* (b *at* $x = b$ *at* y) for every predicate b. Thus predicate values depend only upon computations of component processes and not on the way independent events are ordered in a linear representation of the computation. A predicate c is a *constant* means c *at* $x = c$ *at* y, for all computations x, y. We now define (P *knows* b) *at* x.

Definition: (P *knows* b) *at* $x =$ for all y: $x\,[P]\,y : b$ *at* y

Note that b may itself be a predicate of the form Q *knows* b' in the above definition. We next note some facts about knowledge predicates. In the following, x, y are arbitrary computations, b, b' are arbitrary predicates and P, Q are arbitrary sets of processes. All facts are universally quantified over all computations. We use the convention that P *knows* Q *knows* b *at* x is to be interpreted as (P *knows* (Q *knows* b)) *at* x.

1. P *knows* b *at* $x =$ for all y: $x\,[P]\,y$: P *knows* b *at* y
2. $x\,[P]\,y$ *implies* [P *knows* b *at* x = P *knows* b *at* y]
3. (P *knows* b) *implies* ($P \cup Q$ *knows* b)
4. (P *knows* b) *implies* (b)
5. (P *knows* b) *or* ($\sim P$ *knows* b)
6. (P *knows* b) *and* (P *knows* b') $=$ P *knows* (b *and* b')
7. ((P *knows* b) *or* (P *knows* b')) *implies* (P *knows* (b *or* b'))
8. (P *knows* $\sim b$) *implies* ($\sim P$ *knows* b)
9. ((P *knows* b) *and* (b *implies* b')) *implies* (P *knows* b')
10. P *knows* P *knows* b $=$ P *knows* b
11. P *knows* $\sim P$ *knows* b $=$ $\sim P$ *knows* b
12. P *knows* c, for any constant c.

These facts are easily derivable from the definition of *knows*. We give a proof of (11), whose validity in other domains have been questioned on philosophical grounds [3].

Lemma 2: P *knows* $\sim P$ *knows* b $=$ $\sim P$ *knows* b

□

Proof: P *knows* $\sim P$ *knows* b *at* x
$=$ for all y: $x\,[P]\,y$: $\sim P$ *knows* b *at* y, from definition
$=$ for all y: $x\,[P]\,y$: there exists z: $y\,[P]\,z$: $\sim b$ *at* z, from definition
$=$ there exists z: $x\,[P]\,z$: $\sim b$ *at* z, since $[P]$ is an equivalence relation
$=$ $\sim P$ *knows* b *at* x

□

There are situations where multiple levels of knowledge such as, P *knows* Q *knows* b, are useful. For instance, consider a *token bus* which is a linear sequence of processes among which a token is passed back and forth; processes at the left or right boundary have only a right or left neighbor to whom they may pass the token; other processes may send it to either neighbor. There is only one token in the system and initially it is at the leftmost process. Consider a token bus with five processes labelled p, q, r, s, t from left to right. When r holds the token,

r *knows* ((q *knows* (p does not hold the token)) *and* (s *knows* (t does not hold the token)))

Relations of the form $[\,P\,Q\,]$, with multiple process sets arise from predicates with multiple occurrence of *knows*;

For instance:
p knows q knows b at z
$=$ for all y: $x\,[p]\,y$: *q knows b at y*
$=$ for all y: $x\,[p]\,y$: (for all z: $y\,[q]\,z$: *b at z*)
$=$ for all z: $x\,[p\ q]\,z$: *b at z*

4.2. Local Predicates

Let b be a predicate on system computations, and P a set of processes. We define a predicate *P sure b* as follows.

Definition: (*P sure b*) *at x* $\equiv$ [(*P knows b*) *at x or* (*P knows* $\sim$*b*) *at x*]

In other words (*P sure b*) *at x* means that P knows the value of *b at x*.

We define *unsure* as negation of *sure*.

Definition: *P unsure b* $\equiv$ $\sim$*P sure b*

Hence, (*P unsure b*) *at x* $=$ [($\sim$*P knows b*) *at x and* ($\sim$*P knows* $\sim$*b*) *at x*]

Definition: b is *local* to P $\equiv$ for all x: (*P sure b*) *at x*.

That is, the value of b is *always* known to P. Local predicates capture our intuitive notion of a predicate whose value is controlled by the actions of processes to which it is local.

We note the following facts about local predicates; in the following, b is an arbitrary predicate and P, Q are arbitrary sets of processes.

1. (*b is local to P and* $x\,[P]\,y$) *implies* (*b at x* $=$ *b at y*)
2. *b is local to P implies* (b $=$ *P knows b*)
3. *b is local to P* $=$ ($\sim$*b*) *is local to P*.
4. *b is local to P implies* [*Q knows b* $=$ *Q knows P knows b*]
5. (*P knows b*) *is local to P*.
6. *b is local to P and b is local to Q and P,Q* are disjoint *implies* b is a constant.
7. b is a constant *implies b is local to P*.
8. (*P sure b*) *is local to P*.

Proof of (1) follows from definition of knowledge and local predicates. (2) and (3) follow trivially. (4) follows from *Q knows b at x* $=$ for all y: $x\,[Q]\,y$: *b at y* $=$ for all y : $x\,[Q]\,y$: *P knows b at y* (since b is local to P) $=$ *Q knows P knows b at x*. (5) follows from, (*P knows P knows b* or *P knows* $\sim$*P knows b*) $=$ (*P knows b* or $\sim$*P knows b*) $=$ *true*. Proof of (6) is important and hen e is given below as a lemma. (7) and (8) are trivially proven from definition.

Lemma 3: *b is local to* disjoint sets *P, Q implies* b is a constant

□

Proof: We show that *b at x* $=$ *b at null*, for all x. Proof is by induction on length of x.

b at null $=$ *b at null.*
b at (x;e) $=$ *b at x*, because event e is not on P or e is not on Q, and hence
(x;e) $[P]$ x or (x;e) $[Q]$ x;
then the result follows from property (1).

□

For a system of processes, *b is common knowledge* is defined as the greatest fix point of the following equation.

b is common knowledge $\equiv$ *b and* (*p knows b*) *is common knowledge*, for all processes p. Intuitively, b *is common knowledge* means b is *true*, every process *knows b*, every process *knows* that every process *knows b*, etc.

Halpern and Moses [3] have shown that common knowledge cannot be gained, if it was not present initially, in a system which does not admit of simultaneous events. The following corollary to lemma 3 shows that common knowledge can *be neither gained nor lost* in distributed systems.

Corollary: In a system with more than one process, for any predicate b, *b is common knowledge* is a constant.

□

Proof: For any process p, *b is common knowledge* = *p knows (b is common knowledge)*. Hence, *b is common knowledge* is local to every p. Applying lemma 3, *b is common knowledge* is a constant.

□

It is possible to show that even weaker forms of knowledge cannot be gained or lost in our model of distributed systems. Process sets P, Q have *identical knowledge* of b means,
P knows b = *Q knows b*

Corollary: If P, Q are disjoint and have identical knowledge of b then *P knows b* (and also *Q knows b*) is a constant.

□

Proof: *P knows b* is local to P and *Q knows b* is local to Q. From *P knows b* = *Q knows b*, they are also local to Q and P respectively. The result follows directly from lemma 3.

□

Corollary: If P,Q are disjoint and *P sure b* = *Q sure b*, then *P sure b* (and also *Q sure b*) is a constant.

□

4.3. How Knowledge Is Transferred

We show in this section that chains of knowledge are gained or lost in a sequential manner.

Theorem 4: For arbitrary process sets $P_1, \ldots, P_n$, $n \geq 1$, predicate b and computations x, y,
(P_1 *knows* ... P_n *knows b at x and* $x[P_1 \ldots P_n]y$) *implies* (P_n *knows b at y*)

□

Proof: Proof is by induction on n. For $n = 1$, P_1 *knows b at x*, $x[P_1]y$ *implies* P_1 *knows b at y*, trivially.

Assume the induction hypothesis for some $n - 1$, $n > 1$, and assume

P_1 *knows* ... P_n *knows b at x* and $x[P_1 \ldots P_n]y$.

We shall prove P_n *knows b at y*.

From $x[P_1 \ldots P_n]y$, we conclude that there is a z such that,

$x[P_1 \ldots P_{n-1}]z$ and $z[P_n]y$.

From $x[P_1 \ldots P_{n-1}]z$ and P_1 *knows* ... P_{n-1} *knows* (P_n *knows b*) *at x*, we conclude, using induction, P_{n-1} *knows* P_n *knows b at z*. Hence, P_n *knows b at z*.

Since $z[P_n]y$, P_n *knows b at y*.

□

Corollary: For arbitrary process sets $P_1 \ldots P_n$, $n \geq 1$, predicate b and computations x, y,

(P_1 *knows* ... P_{n-1} *knows* $\sim P_n$ *knows b at x and* $x[P_1 \ldots P_n]y$) *implies* $\sim P_n$ *knows b at y*

□

Note: For $n = 1$ antecedant is, $\sim P_n$ *knows b at x*.

Corollary: Theorem 4 holds with *knows* replaced by *sure*.

Theorem 4 can be applied to (1) $x \leq y$ (knowledge is lost) and (2) $y \leq x$ (knowledge is gained). Using theorem 1, we can deduce that there is a process chain $< P_1 \ldots P_n >$ in the former case and $< P_n \ldots P_1 >$ in the latter case. We first prove a simple lemma about the effect of receive or send on knowledge: we show that certain forms of knowledge cannot be lost by receiving nor gained by sending.

Lemma 4: (How events at a process change its knowledge)

Let b be a predicate which is local to $\overline{P}$ and $(x;e)$ a computation where e is an event on P.

1. e is a receive: {knowledge is not lost}
   (P *knows b at x*) *implies* (P *knows b at* $(x;e)$)

2. e is a send: {knowledge is not gained}
   (P *knows* b *at* $(x;e)$) *implies* (P *knows* b *at* x)

3. e is an internal event: {knowledge is neither lost nor gained}
   (P *knows* b *at* x) $\equiv$ (P *knows* b *at* $(x;e)$)

$\square$

Proof: We prove only (1). Consider any z such that $(x;e)\ [P]\ z$. We will show b *at* z and hence it follows that P *knows* b *at* $(x;e)$.

Since $z\ [\overline{P}]\ z$, we have $(x;e)\ [P\overline{P}]\ z$.

From theorem 3, since e is a receive, $x\ [P\overline{P}]\ z$. Since b is local to $\overline{P}$,

P *knows* $b \equiv P$ *knows* $\overline{P}$ *knows* b.

From theorem 4,

(P *knows* $\overline{P}$ *knows* b *at* x, $x\ [P\overline{P}]\ z$) *implies*

($\overline{P}$ *knows* b *at* z)

($\overline{P}$ *knows* b *at* z) *implies* (b *at* z)

This completes the proof.

$\square$

Corollary: (b is local to $\overline{P}$, $\sim P$ *knows* b *at* x, P *knows* b *at* y, $x \le y$) *implies* (P receives a message in (x, y)).

$\square$

Corollary: (b is local to $\overline{P}$, P *knows* b *at* x, $\sim P$ *knows* b *at* y, $x \le y$) *implies* (P sends a message in (x, y)).

$\square$

Theorem 5: (How Knowledge Is Gained:) Let x, y be computations where $x \le y$, $\sim(P_n$ *knows* $b)$ *at* x and $(P_1$ *knows* $\ldots$ P_n *knows* $b)$ *at* y, for arbitrary process sets $P_1 \ldots P_n$, $n \ge 1$. Then there is a process chain $<P_n \ldots P_1>$ in (x, y). Furthermore, if b is local to $\overline{P}_n$ then P_n has a receive event in (x, y) such that b *at* z holds for every prefix z of y which includes the corresponding send event.

$\square$

Theorem 6: (How Knowledge Is Lost:)

Let x, y be computations where $x \le y$, P_1 *knows* $\ldots P_n$ *knows* b *at* x and $\sim P_n$ *knows* b *at* y, for arbitrary process sets $P_1 \ldots P_n$, $n \ge 1$. Then there is a process chain $<P_1 \ldots P_n>$ in (x, y). Furthermore, if b is local to $\overline{P}_n$ then P_n has a send event in (x, y).

$\square$

Observe that the statements of the two theorems are not entirely symmetric for receive and send events. The reason is that every computation including a receive must also include the corresponding send, but not conversely.

Theorems 4, 5, 6 and their corollaries hold with *knows* replaced by *sure*.

5. Applications Of The Results

We sketch a few applications of the theory developed so far. A full treatment of these results may be found in [8].

We show that it is impossible for process P to track the change in value of a local predicate of $\overline{P}$, exactly at all times; P must be unsure about the value of this predicate while it is undergoing change. We also show that necessary condition for changing a local predicate b of $\overline{P}$, is that $\overline{P}$ *knows* P *unsure* b, at the point of change.

Traditional techniques for process failure detection based on time-outs assume certain execution speeds for processes and maximum delays for message transfer. It is generally accepted that detection of failure is impossible without using time-outs, a fact that we prove formally. We use the fact that failure of a process is local to the process and the process does not send messages after its failure; hence other processes remain unsure at all points about a process failure.

We show that any algorithm, which detects termination of an underlying computation, requires at least as many overhead messages, in general, for detection as there are messages in the underlying computation. We first show that in order for termination to be detected, an overhead message is sent by some process, without its first receiving a message, after the underlying computation terminates; this fact is proven directly from the theorem of knowledge gain, because detecting termination amounts to gaining knowledge.

Next we show that a process is sometimes required to send an overhead message even when the underlying computation has not terminated, because the computation may be isomorphic (with respect to this process) to a computation in which the underlying computation has terminated. Using these two results, we construct a computation, in which the number of overhead messages is at least as many as the number of underlying messages.

6. Discussion

We have shown that isomorphisms between system computations with respect to a process is a useful concept in reasoning about distributed systems. Isomorphism forms the basis for defining and deriving properties about knowledge. "Scenarios" have been used [7] to show impossibility of solving certain problems; in our context, a scenario is a computation, and isomorphism is the formal treatment of equivalence between scenarios. Theorems on knowledge transfer provide lower bounds on numbers of messages required to solve certain problems. We have used isomorphism as the basis of fusion theorem and related isomorphism to semantics of send, receive and internal events.

A number of generalizations of this work are possible: we can define isomorphism based on states of processes, rather than computations; we can introduce the notion of time into computations; we can define *belief* in terms of isomorphism. Most of the results in this paper are applicable in the first case but not in the other two cases.

Acknowledgement: We are indebted to Shmuel Katz, Joe Halpern, E.W. Dijkstra and Bengt Jonsson for their comments. Particular thanks go to Ernie Cohen for a careful reading of the manuscript and insightful comments.

This work was supported in part by a grant from the Office of Naval Research under N00014-85-K-0057.

REFERENCES

1. K. M. Chandy & J. Misra: "Drinking Philosophers Problem", *TOPLAS*, October 1984.

2. M. J. Fischer, N. Lynch & M. Paterson, "Impossibility of Distributed Consensus with one Faulty Process", *Journal of the ACM*, April 1985.

3. J. Y. Halpern & Y. Moses: "(Knowledge And Common Knowledge In A Distributed Environment", *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Vancouver, Canada, August 1984.

4. J. Hintikka: "*Knowledge and Belief*", *Cornell University Press*, 1962.

5. L. Lamport:, "Time, Clocks and the Orderings of Events in a Distributed System", *Communications of the ACM*, Vol. 21, No. 7, pp. 558-564, July 1978.

6. D. Lehmann, "Knowledge, Common Knowledge, and Related Puzzles", *ACM SIGACT-SIGOPS Symposium of Principles of Distributed Computing*, Vancouver, Canada, August 1984.

7. N. Lynch & M. Fischer, "A Lower Bound for the Time to Assure Interactive Consistency", *Information Processing Letters*, Vol. 14, No. 4, June 1982.

8. K. M. Chandy & Jayadev Misra, "How Processes Learn", Distributed Computing, Vol. 1, No. 1, January 1986, (Published by Springer Verlag).

Distributed Computing (1986) 1:40-52

DISTRIBUTED COMPUTING
© Springer-Verlag 1986

How processes learn

K.M. Chandy and Jayadev Misra
Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712, USA

Jayadev Misra is a professor in the Department of Computer Sciences at the University of Texas at Austin. His primary research interests are in the area of distributed computing: specification and design of networks of asynchronous components. He believes that sound practical techniques must be based on elegant theories.

Mani Chandy is a professor of Computer Science and Electrical Engineering at the University of Texas at Austin. He is chairman of the Computer Sciences Department. His research interests are in distributed systems and performance analysis.

1 Introduction

Processes in distributed systems communicate with one another exclusively by sending and receiving messages. A process has access to its state but not to the states of other processes. Many distributed algorithms require that a process determine facts about the overall system computation. In anthropomorphic terms, processes "learn" about states of other processes in the evolution of system computation. This paper is concerned with how processes learn. We give a precise characterization of the minimum information flow necessary for a process to determine specific facts about the system.

The central concept in our study is that of *isomorphism* between system computations with respect to a process: two system computations are isomorphic with respect to a process if the process behavior is identical in both. In anthropomorphic terms, "system computations are isomorphic with respect to a process" means the process cannot distinguish between them.

Many correctness arguments about distributed systems have the following operational flavor: "I will send a message to you and then you will think that I am busy and so you will broadcast ...". Such operational arguments are difficult to understand and error prone. The basis for such operational arguments is usually a "process chain": a sequence of message transfers along a chain of processes. We advocate nonoperational reasoning. The basis for nonoperational arguments is isomorphism; we relate isomorphism to process chains. Algebraic properties of system computations under isomorphism provide a precise framework for correctness arguments.

It has been proposed [3, 6] that a notion of "knowledge" is useful in studying distributed computations. In earlier works, knowledge is introduced via a set of axioms [4]. Our definition of knowledge is based on isomorphism. Our model allows us to study how knowledge

Offprint requests to: K.M.Chandy

This work was supported in part by a grant from the Office of Naval Research under N00014-85-K-0057

is "gained" or "lost". One of our key theorems states that knowledge gain and knowledge loss both require sequential transfer of information: if process q does not know fact b and later, p knows that q knows b, then q must have communicated with p, perhaps indirectly through other processes, between these two points in the computation; conversely, if p knows that q knows b and later, q does not know b then p must have communicated with q between these two points in the computation. In the first case, the effect of communication is to inform p of q's knowledge of b. Analogously, in the second case, the effect of communication is to inform q of p's intention of relinquishing its knowledge (that q knows b). Generalizations of these results for arbitrary sequences of processes are stated and proved as corollaries of a general theorem on isomorphism.

We use the results alluded to in the last paragraph for proving lower bounds on the number of messages required to solve certain problems. We show, for instance, that there is no algorithm to detect termination of an underlying computation using only a bounded number of overhead messages.

2 Model of a distributed system

A distributed system consists of a finite set of processes. A process is characterized by a set of process computations each of which is a finite sequence of events on that process. Process computations are prefix closed, i.e. all prefixes of a process computation are also process computations (of that process). An event on a process is either a *send*, a *receive* or an *internal event*. A *send* event on a process corresponds to sending a message to another process. A *receive* event on a process corresponds to reception of a message by the process. There is no external communication associated with an *internal* event. For a set of processes P, a send event by P is a send event by some component process of P to a process outside P; similarly a receive event by P denotes receipt by some process in P of a message sent from outside P. Communication among processes in P are internal events of P. We use "e is on P", for event e and process set P, to denote that e is an event on some process in P. We rule out processes which have no event in any computation. We assume that all events and all messages are distinguished; for instance, multiple occurrences of the same message are distinguished by affixing sequence numbers to them.

Let z be any sequence of events on component processes of a distributed system. The *projection* of z on a component process p, denoted by z_p, is the subsequence of z consisting of all events on p. A finite sequence of events z is a *system computation* of a distributed system means (1) for all processes p, z_p is a process computation of p and, (2) for every receive event in z, say receipt of message m by process p, there is a send event, of sending m to p, which occurs earlier than the receive in z: this send event will be called the send event *corresponding* to the receive. We leave it to the reader to show that system computations are prefix closed.

In this paper we consider a single (generic) distributed system. For instance, when we say "z is a computation" we mean that z is a computation of the distributed system considered here. We use *computation* to mean *system computation* when no confusion can arise.

Notation. We use x, y, z to denote computations, p, q for processes and P, Q for process sets; these symbols may be used with subscripts or superscripts. The concatenation of two sequences y and z will be denoted by $(y; z)$. For sequences y and z, $y \leqq z$ denotes that y is a prefix of z; in this case (y, z) denotes the suffix of z obtained by removing y from z. The empty sequence will be denoted by *null*. The symbol $=$ is used to denote equalities among sets and among predicates. The symbol $\equiv$ is used for definitions. The set of all processes in the system will be denoted by D and for any process set P, $\bar{P} = D - P$.

3 Isomorphism

We define relation $[p]$ on the set of system computations as follows.

Definition. For system computations x, y: $x[p]y \equiv (x_p = y_p)$.

In other words, $x[p]y$ means p's computation is the same in system computations x and y. In this case, we say x, y are *isomorphic with respect to p*. For a process set P, define relation $[P]$, on the system computations, as follows.

Definition. $x[P]y \equiv$ for all p in P, $x[p]y$.

Thus $x[P]y$ means that, given only the computations of processes in P we cannot distinguish x from y. From definition, $x[\{\ \}]y$, for all computations x, y where $\{\ \}$ denotes the empty set. Observe that $[P]$ is an equivalence relation.

It is convenient to represent all such isomorphism relations by an *isomorphism diagram:* an undirected labelled graph whose vertices are computations and there is an edge labelled $[P]$ between vertices x, y if P is the largest set of processes for which $x[P]y$. Observe that every vertex has a self loop labelled $[D]$ where D is the set of all processes in the system. Note that $x[D]y$, $x \neq y$, implies y is a permutation of x.

Example 1. Consider a system with two processes, p *and* q, for which part of the isomorphism diagram, showing the relationships among four system computation, is given below.

From the diagram $x[p]y$, but not $x[q]y$. This means p has the same computations in both x and y, whereas q's computations in x and y differ. Computations x and z have the same computations for both p and q; hence one is a permutation of the other. There is no direct relationship between y and w; neither $y[p]w$ nor $y[q]w$ holds. However, there is an indirect relationship between y and w because $y[p]z$ and $z[q]w$. We explore such indirect relationships next. □

Definition. Let $n>0$ and P_i be process sets, $0 \leq i \leq n$.

$x[P_0 \dots P_n]z \equiv x[P_0 \dots P_{n-1}]y$ *and* $y[P_n]z$, for some computation y.

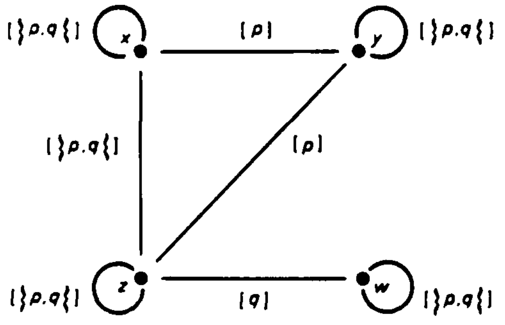


Fig. 1. An isomorphism diagram

Hence, $[PQ]=[P]\circ[Q]$ where "$\circ$" is the relational composition operator. This operator is associative (from properties of relations). In terms of the isomorphism diagram, $x[P_0 \dots P_n]z$ means there is a path from x to z whose edges are labelled with $Q_0, \dots, Q_n$, respectively, where $Q_i \supseteq P_i$, for all i.

Example 1 (contd.). We have $y[pq]w$ and $w[qp]y$. Also, trivially, $y[qp]z$, $y[qpq]z$, etc. □

We note some properties of isomorphism relations. In the following, P, $P_1, \dots, P_n$, Q, denote arbitrary process sets and x, y, z denote arbitrary computations.

1. $[P]$ is an equivalence relation.
2. (Substitution) $([\beta]=[\delta])$ *implies* $([\alpha\beta\gamma]=[\alpha\delta\gamma])$ for arbitrary sequences of process sets α, β, γ, δ.
3. (Idempotence) $[PP]=[P]$
4. (Reflexivity) $x[P_1 \dots P_n]x$
5. (Inversion) $x[P_1 \dots P_n]y = y[P_n \dots P_1]x$
6. (Concatenation) For $0<m<n$, $\exists y$: $x[P_1 \dots P_m]y$, $y[P_{m+1} \dots P_n]z = x[P_1 \dots P_m P_{m+1} \dots P_n]z$
7. $[P \cup Q]=([P] \cap [Q])$
8. $(Q \supseteq P)=([Q] \subseteq [P])$
9. $(P=Q)=([P]=[Q])$
10. $Q \supseteq P$ *implies* $([QP]=[P]=[PQ])$

These properties follow from properties of relations and our model. We only sketch a proof of one part of property 8:

$([Q] \subseteq [P])$ *implies* $(Q \supseteq P)$.

If $Q \not\supseteq P$ then there is a process p in $P-Q$. From our model, p has an event e in some computation $(x;e)$. Then $x[Q](x;e)$ and $\sim x[P](x;e)$. Hence $[Q] \not\subseteq [P]$.

3.1 Process chains

As noted in the introduction, the basis for many operational arguments are process chains: process p informing q which in turn, informs r etc. One of our goals is to replace such concepts by algebraic properties of system computations. In this section we show how process chains are related to isomorphism. We first define proces chains; this definition is along the lines suggested by Lamport [5].

Definition. For events e, e' in a computation z, $e \xrightarrow{z} e'$ means:

1. e' is a receive and e is the corresponding send, or
2. events e, e' are in the same process computation and ($e=e'$ or e occurs earlier than e'), or
3. there exists an event e'' such that $e \xrightarrow{z} e''$ and $e'' \xrightarrow{z} e'$.

For brevity we write $e \rightarrow e'$ when the computation z is understood from context. We will write $e_0 \rightarrow e_1 \rightarrow \dots e_{n-1} \rightarrow e_n$, as shorthand for $e_0 \rightarrow e_1$ and ... and $e_{n-1} \rightarrow e_n$. Observe that $e \rightarrow e$ for every event e in z. A computation z has a *process chain* $\langle P_0 P_1 \dots P_n \rangle$ means there exist events $e_0, e_1, \dots e_n$, not necessarily distinct, in z such that event e_i is on P_i, for all $0 \leq i \leq n$, and $e_0 \rightarrow e_1 \rightarrow \dots \rightarrow e_n$.

Observation 1. Any occurrence of "P" in a process chain may be replaced by "PP", or vice versa, since for any event e on P, $e \rightarrow e$.

Observation 2. Let x be a sequence consisting of a subset of events from a computation y. Suppose that for every event e in x: every e', where $e' \xrightarrow{y} e$, is also in x, and $e' \xrightarrow{x} e$. Then x is a computation.

3.2 Relationship between isomorphism and process chain

Theorem 1. (Fundamental theorem of process chains). *Let z be a computation and x a prefix of z. Let $P_1, P_2 \dots P_n$, $n \geq 1$, be sets of processes. Then $x[P_1 P_2 \dots P_n]z$ or there is a process chain $\langle P_1 P_2 \dots P_n \rangle$ in (x,z).* ☐

Proof. Assuming that there is no process chain $\langle P_1 \dots P_n \rangle$ in (x,z), we show that $x[P_1 \dots P_n]z$. Proof is by induction on n. For $n=1$, absence of a process chain $\langle P_1 \rangle$ in (x,z) means that there is no event on P_1 in (x,z) and hence $x[P_1]z$. For $n>1$, we show that there is some y, $x \leq y$, such that there is no process chain $\langle P_1 \dots P_{n-1} \rangle$ in (x,y) and $y[P_n]z$; the result then follows by inductive argument.

Let E be the subsequence of events in (x,z) consisting of the set of events $\{e | e \rightarrow e'$ where e is in (x,z) and e' is some event on $P_n\}$. Let $y = (x; E)$. First, we show that if $e_1 \xrightarrow{z} e_2$ and e_2 is in y then e_1 is also in y and $e_1 \xrightarrow{y} e_2$; this guarantees (from observation 2) that y is a computation. This result follows trivially when e_2 is in x. If e_2 is in (x,z) then $e_2 \xrightarrow{z} e'$, for some event e' on P_n and hence $e_1 \xrightarrow{z} e'$ and therefore e_1 is in y; the relative order between e_1, e_2 is maintained by our construction.

Next, we show that $y[P_n]z$; that is, every event on P_n that is in z is also in y. This follows trivially for events on P_n that are in x. Let e' be an event on P_n that is in (x,z). Since $e' \rightarrow e'$, e' is also in E and hence in y.

Finally, we show that there is no process chain $\langle P_1 \dots P_{n-1} \rangle$ in (x,y). If there is such a process chain, consider its last event e. According to our construction, event chain $e \rightarrow e'$ exists in (x,z), where e' is some event on P_n. Hence there is a process chain $\langle P_1 \dots P_n \rangle$ in (x,z), contradicting our assumption. ☐

We note that the two conditions in the last sentence of the theorem are not exclusive. Consider two computations z, z' where

z is $\langle P$ sends m to R; R receives m from P; R sends m' to Q; Q receives m' from $R \rangle$,

z' is $\langle R$ sends m' to Q: Q receives m' from $R \rangle$

In z, though there is a process chain $\langle PRQ \rangle$, there is not a "true" dependence from P to R to Q: R sends m' to Q *independent* of receiving m from P (as shown in z'). Note that *null* $[P]$ z' and z' $[Q]$ z, and hence *null* $[PQ]$ z, though $(null, z)$ has a process chain $\langle PQ \rangle$.

3.3 An application of isomorphism: how to construct a computation by fusing separate ones

In this section, we show an application of isomorphism: we give a construction to "fuse" two computations to obtain a new computation, provided certain types of paths exist in the isomorphism diagram. We motivate the discussion by the following observations. Suppose $(x; E)$ and $(x; \bar{E})$ are computations where all events in E are on a process set P and all events in $\bar{E}$ are on $\bar{P}$. Then, from definition, $(x; \bar{E}; E)$ and $(x; E; \bar{E})$ are also computations, because events in $E, \bar{E}$ are independent and hence may be fused in arbitrary order. A similar result appears in Fischer et al. [2]. The following lemma is a generalization of this obervation.

Lemma 1. *Let x, y, z be computations where $x \leq y$ and $x \leq z$. Let P, Q be such that $P \cup Q = D$, $x[P]y$ and $x[Q]z$. Then there exists a computation w where $x \leq w$, $y[Q]w$ and $z[P]w$.* ☐

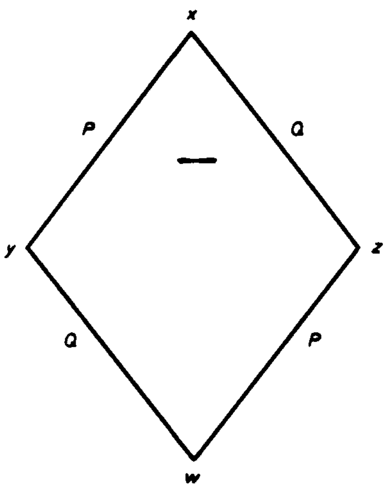


Fig. 2. Isomorphism diagram depicting fusion

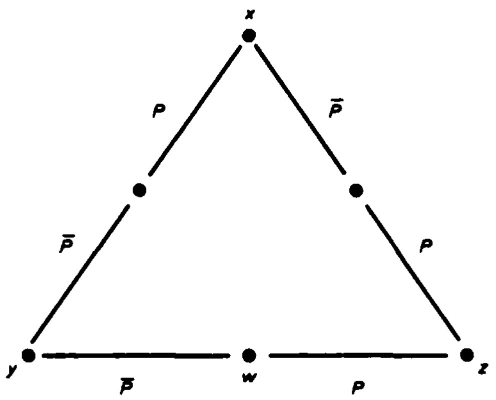


Fig. 3. Diagramatic representation of fusion theorem

The relationships among x, y, z and w are represented by the following commutative isomorphism diagram.

Proof. Let $w = x; (x, y); (x, z)$.

From the condition of the lemma, (x, y) has events only on $\bar P$ and (x, z) has events only on $\bar Q$. Since $P \cup Q = D$, $\bar P \cap \bar Q = \{\ \}$ and hence no process has events in both (x, y) and (x, z). It follows, from definition of computations, that w is a computation. Also $y[Q]w$, $z[P]w$ and $x \leqq w$, as required for proof of the lemma. □

Note that, in the construction of Lemma 1, all events from E and $\bar E$ were present in the fused computation. We prove a far more general result below. We show that for any two arbitrary computations y and z, the projected computations, y_P and z_P, may be fused to form a new computation provided there is a computation x which is a prefix of both y and z, and no message sent by $\bar P$ in (x, y) is received by P in (x, y) and no message sent by P in (x, z) is received by $\bar P$ in (x, z). This makes intuitive sense: processes in P can execute all events in y given only that processes in $\bar P$ execute all events up to x and similarly for executions of events on $\bar P$ up to z. However, the statement and proof of this result are difficult without the notion of isomorphism. We note that the result may be easily generalized to fusions of arbitrary numbers of computations under similar constraints.

Theorem 2. (Fusion of computations). *Consider system computations x, y, z where $x \leqq y$ and $x \leqq z$. Let P be a set of processes such that there is no process chain, (1) $\langle P \bar P \rangle$ in (x, y) and (2) $\langle \bar P P \rangle$ in (x, z). Then there is a computation w where, $x \leqq w$, $y[\bar P]w$ and $z[P]w$. That is, w consists of all events on $\bar P$ from y and all events on P from z.* □

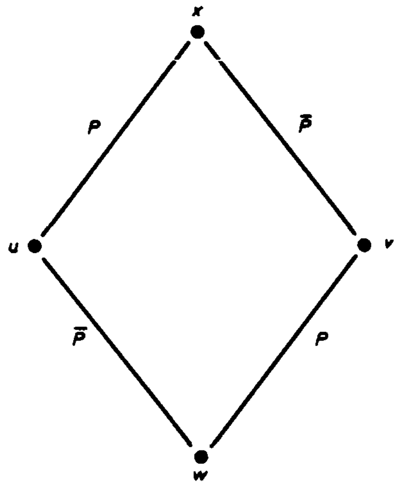


Fig. 4. Intermediate step in fusion theorem

Proof. According to Theorem 1, absence of process chains as given in this theorem means that, $x[P \bar P]y$ and $x[\bar P P]z$.

The theorem asserts the existence of the isomorphism diagram in Fig. 3. To prove that such a w exists, label the intermediate point between x, y as u and between x, z as v in this figure. Now we apply Lemma 1 to x, u, v to obtain a w, as given in Fig. 4.

Now $u[\bar P]y$ and $u[\bar P]w$; hence $y[\bar P]w$. Similarly $z[P]w$. This proves the theorem. Relationships among x, y, z, u, v, w are shown in Fig. 5. □

The fusion theorem is used later to obtain lower bounds on the number of messages required to solve certain problems.

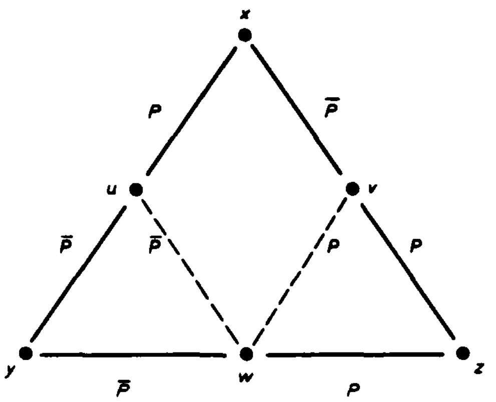


Fig. 5. Isomorphism diagram depicting proof of fusion theorem

3.4 *Semantics of event types in terms of isomorphism*

We now use isomorphism to state and derive some important facts about various types of events. First, note that a process carries out an internal event or sends a message depending on its own computation alone. Therefore, if a process takes such a step in a computation x, it will also do so in y, if x, y are isomorphic with respect to this process. An analogous result holds for internal and receive events. The following principle, which states these facts formally, may be proven from the definition of system computation.

Principle of computation extension:

Let e be an event on P.

1. e is an internal or send event: $(x[P]y$ *and* $(x;e)$ is a computation) *implies* $(y;e)$ is a computation.
2. e is an internal or receive event: $(x;e)[P]y$ *implies* $(y-e)$ is a computation, where $(y-e)$ is the sequence obtained by deleting e from y. □

Note. In (1), $(x;e)[P]$ $(y;e)$ and in (2), $x[P](y-e)$.

Corollary. *Let e be a receive event on P and let the corresponding send event be on Q.*

$(x[P\cup Q]y$ *and* $(x;e)$ *is a computation) implies* $(y;e)$ *is a computation.* □

Proof. e is an internal event of $P\cup Q$. □

The following theorem follows from the principle of computation extension.

Theorem 3. *Let* $(x;e)$ *be a computation where e is an event on P.*

Case 1. e is a receive:

for every z: $(x;e)$ $[P\bar{P}]z$ *implies* $x[P\bar{P}]z$

Case 2. e is a send:

for every z: $x[P\bar{P}]$ z *implies* $(x;e)[P\bar{P}]z$

Case 3. e is an internal event:

for every z: $(x;e)[P\bar{P}]z = x[P\bar{P}]z$ □

Proof. We will prove only Case 2: other cases are similarly proven.

$x[P\bar{P}]z$ *implies* there exists y, $x[P]y$ *and* $y[\bar{P}]z$.

From principle of computation extension, $(y;e)$ is a computation and $(x;e)$ $[P]$ $(y;e)$. Also, $(y;e)[\bar{P}]y$. Hence, $(x;e)[P\bar{P}]z$, and therefore, $(x;e)$ $[P\bar{P}]z$. □

This theorem captures the intuitive notion that the set of possible computations, isomorphic with respect to P, can only shrink in size as a result of a reception as computations which do not include the corresponding send are ruled out. Similarly, the set of possible computations, isomorphic with respect to P cannot shrink as a result of a send: after the send, additional computations which accept the message sent are isomorphic while all prior isomorphic computations remain isomorphic. An internal event can neither expand nor shrink the set of isomorphic computations.

4 Knowledge

As we have remarked earlier, predicates of the type *P knows b at x* may be defined using isomorphism. We explore properties of such predicates in our model. We show that they satisfy the "knowledge axioms" as given in [3, 6]. We prove a general result which shows that certain forms of knowledge can only be gained or lost in a sequential fashion along a chain of processes. That is, if b is *false* for a computation and later, P_1 *knows* P_2 *knows* ... P_n *knows b* (this represents knowledge gain), then there is a process chain $\langle P_n P_{n-1} \dots P_1\rangle$ between these two points of the computation. Conversely, if P_1 *knows* P_2 *knows* ... P_n *knows b* and later, b is *false* (this represents knowledge loss), then there is a process chain $\langle P_1 P_2 \dots P_n\rangle$ between these two points of the computation.

Crucial to our work is the notion of *local predicates*: a predicate local to p can change in value only as a result of events on p. We show that local predicates play a key role in understanding knowledge predicates.

4.1 Knowledge predicates

Let b denote a predicate on system computations and "b *at* x" its value for computation x. Our predicates are *total*, i.e. for each x, b *at* x is either *true* or *false*. We furthermore assume that $x[D]y$ *implies* (b *at* $x = b$ *at* y) for every predicate b. Thus predicate values depend only upon computations of component processes and not on the way independent events are ordered in a linear representation of the computation. A predicate c is a *constant* means c *at* $x = c$ *at* y, for all computations x, y. We now define (P *knows* b) *at* x.

Definition. (P *knows* b) *at* x =
for all y: $x[P]y$: b *at* y

Note that b may itself be a predicate of the form Q *knows* b' in the above definition. We next note some facts about knowledge predicates. In the following, x, y are arbitrary computations, b, b' are arbitrary predicates and P, Q are arbitrary sets of processes. All facts are universally quantified over all computations. We use the convention that P *knows* Q *knows* b *at* x is to be interpreted as (P *knows* (Q *knows* b)) *at* x.

1. P *knows* b *at* x = for all y: $x[P]y$: P *knows* b *at* y
2. $x[P]y$ *implies* [P *knows* b *at* x = P *knows* b *at* y]
3. (P *knows* b) *implies* ($P \cup Q$ *knows* b)
4. (P *knows* b) *implies* (b)
5. (P *knows* b) *or* ($\sim P$ *knows* b)
6. (P *knows* b) *and* (P *knows* b') = P *knows* (b *and* b')
7. ((P *knows* b) *or* (P *knows* b')) *implies* (P *knows* (b *or* b'))
8. (P *knows* $\sim b$) *implies* ($\sim P$ *knows* b)
9. ((P *knows* b) *and* (b *implies* b')) *implies* (P *knows* b')
10. P *knows* P *knows* b = P *knows* b
11. P *knows* $\sim P$ *knows* b = $\sim P$ *knows* b
12. P *knows* c *or* P *knows* $\sim c$, for any constant c.

These facts are easily derivable from the definition of *knows*. We give a proof of (11), whose validity in other domains have been questioned on philosophical grounds [3].

Lemma 2. P *knows* $\sim P$ *knows* $b = \sim P$ *knows* b. □

Proof. P *knows* $\sim P$ *knows* b *at* x
= for all y: $x[P]y$: $\sim P$ *knows* b *at* y,
from definition
= for all y: $x[P]y$: there exists z: $y[P]z$: $\sim b$ *at* z,
from definition
= there exists z: $x[P]z$: $\sim b$ *at* z,
since $[P]$ is an equivalence relation
= $\sim P$ *knows* b *at* x. □

There are situations where multiple levels of knowledge such as, P *knows* Q *knows* b, are useful. For instance, consider a *token bus* which is a linear sequence of processes among which a token is passed back and forth; processes at the left or right boundary have only a right or left neighbor to whom they may pass the token; other processes may send it to either neighbor. There is only one token in the system and initially it is at the leftmost process. Consider a token bus with five processes labelled p, q, r, s, t from left to right. When r holds the token,

r *knows* ((q *knows* (p does not hold the token)) *and*
(s *knows* (t does not hold the token)))

Relations of the form $[PQ]$, with multiple process sets, arise from predicates with multiple occurrence of *knows*;
For instance:

p *knows* q *knows* b *at* z
= for all y: $x[p]y$: q *knows* b *at* y
= for all y: $x[p]y$: (for all z: $y[q]z$: b *at* z)
= for all z: $x[pq]z$: b *at* z

4.2 Local predicates

Let b be a predicate on system computations, and P a set of processes. We define a predicate P *sure* b as follows.

Definition. (P *sure* b) *at* $x \equiv$ ((P *knows* b) *at* x *or* (P *knows* $\sim b$) *at* x).

In other words (P *sure* b) *at* x means that P knows the value of b *at* x.

We define *unsure* as negation of *sure*.

Definition. P *unsure* $b \equiv \sim P$ *sure* b.

Hence, (P *unsure* b) *at* x = [($\sim P$ *knows* b) *at* x *and* ($\sim P$ *knows* $\sim b$) *at* x].

Definition. b is *local* to $P \equiv$ for all x: (P *sure* b) *at* x.

That is, the value of b is *always* known to P. Local predicates capture our intuitive notion of a predicate whose value is controlled by the actions of processes to which it is local.

We note the following facts about local predicates; in the following, b is an arbitrary predicate and P, Q are arbitrary sets of processes.

1. (b *is local to* P *and* $x[P]y$) *implies* (b *at* $x = b$ *at* y)
2. b *is local to* P *implies* ($b = P$ *knows* b)
3. b *is local to* $P = (\sim b)$ *is local to* P.
4. b *is local to* P *implies* [Q *knows* $b = Q$ *knows* P *knows* b]
5. (P *knows* b) *is local to* P.
6. b *is local to* P *and* b *is local to* Q *and* P, Q are disjoint *implies* b is a constant.
7. b is a constant *implies* b *is local to* P.
8. (P *sure* b) *is local to* P.

Proof of (1) follows from definition of knowledge and local predicates. (2) and (3) follow trivially. (4) follows from Q *knows* b *at* $x =$ for all y: $x[Q]y$: b *at* $y =$ for all y: $x[Q]y$: P *knows* b *at* y (since b is local to P) $= Q$ *knows* P *knows* b *at* x. (5) follows from, (P *knows* P *knows* b or P *knows* $\sim P$ *knows* b) $=$ (P *knows* b or $\sim P$ *knows* b) $=$ *true*. Proof of (6) is important and hence is given below as a lemma. (7) and (8) are trivially proven from definition.

Lemma 3. *b is local to disjoint sets P, Q implies b is a constant.* □

Proof. We show that b *at* $x = b$ *at null*, for all x. Proof is by induction on length of x.

b *at null* $= b$ *at null*.
b *at* $(x;e) = b$ *at* x, because event e is not on P or e is not on Q, and hence $(x;e)[P]x$ or $(x;e)[Q]x$; then the result follows from property (1). □

For a system of processes, *b is common knowledge* is defined as the greatest fix point of the following equation.

b is common knowledge $\equiv$ *b and (p knows b) is common knowledge*, for all processes p. Intuitively, *b is common knowledge* means b is *true*, every process *knows* b, every process *knows* that every process *knows* b, etc.

Halpern and Moses [3] have shown that common knowledge cannot be gained, if it was not present initially, in a system which does not admit of simultaneous events. The following corollary to lemma 3 shows that common knowledge can *neither be gained nor lost* in distributed systems.

Corollary. *In a system with more than one process, for any predicate b, b is common knowledge is a constant.* □

Proof. For any process p, *b is common knowledge* $=$ *p knows (b is common knowledge)*. Hence, *b is common knowledge* is local to every p. Applying lemma 3, *b is common knowledge* is a constant. □

It is possible to show that even weaker forms of knowledge cannot be gained or lost in our model of distributed systems. Process sets P, Q have *identical knowledge* of b means.

P *knows* $b = Q$ *knows* b

Corollary. *If P, Q are disjoint and have identical knowledge of b then P knows b (and also Q knows b) is a constant.* □

Proof. P *knows* b is local to P and Q *knows* b is local to Q. From P *knows* $b = Q$ *knows* b, they are also local to Q and P respectively. The result follows directly from lemma 3. □

Corollary. *If P, Q are disjoint and P sure $b = Q$ sure b, then P sure b (and also Q sure b) is a constant.* □

4.3 How knowledge is transferred

We show in this section that chains of knowledge are gained or lost in a sequential manner.

Theorem 4. *For arbitrary process sets $P_1, \dots, P_n$, $n \geq 1$, predicate b and computations x, y,*

(P_1 *knows* ... P_n *knows* b *at* x *and* $x[P_1 \dots P_n]y$) *implies* (P_n *knows* b *at* y). □

Proof. Proof is by induction on n. For $n = 1$, P_1 *knows* b *at* x, $x[P_1]y$ *implies* P_1 *knows* b *at* y, trivially.

Assume the induction hypothesis for some $n-1$, $n > 1$, and assume

P_1 *knows* ... P_n *knows* b *at* x and $x[P_1 \dots P_n]y$.

We shall prove P_n *knows* b *at* y.

From $x[P_1 \dots P_n]y$, we conclude that there is a z such that,

$x[P_1 \dots P_{n-1}]z$ and $z[P_n]y$.

From $x[P_1 \ldots P_{n-1}]z$ and P_1 *knows* ... P_{n-1} *knows* (P_n *knows* b) *at* x, we conclude, using induction, P_{n-1} *knows* P_n *knows* b *at* z. Hence, P_n *knows* b *at* z.

Since $z\,[P_n]\,y$, P_n *knows* b *at* y. □

Corollary. *For arbitrary process sets* $P_1 \ldots P_n$, $n \geqq 1$, *predicate* b *and computations* x, y,

(P_1 *knows* ... P_{n-1} *knows* $\sim P_n$ *knows* b *at* x *and* $x[P_1 \ldots P_n]\,y$) *implies* $\sim P_n$ *knows* b *at* y. □

Note. For $n=1$ antecedant is, $\sim P_n$ *knows* b *at* x.

Corollary. *Theorem* 4 *holds with knows replaced by sure in* "P_n *knows*".

Theorem 4 can be applied to (1) $x \leqq y$ (knowledge is lost) and (2) $y \leqq x$ (knowledge is gained). Using theorem 1, we can deduce that there is a process chain $\langle P_1 \ldots P_n \rangle$ in the former case and $\langle P_n \ldots P_1 \rangle$ in the latter case. We first prove a simple lemma about the effect of receive or send on knowledge: we show that certain forms of knowledge cannot be lost by receiving nor gained by sending.

Lemma 4. *(How events at a process change its knowledge)*

Let b *be a predicate which is local to* $\bar{P}$ *and* $(x;e)$ *a computation where* e *is an event on* P.

1. *e is a receive:* {*knowledge is not lost*}
   (P *knows* b *at* x) *implies* (P *knows* b *at* $(x;e)$)
2. *e is a send:* {*knowledge is not gained*}
   (P *knows* b *at* $(x;e)$) *implies* (P *knows* b *at* x)
3. *e is an internal event:*
   {*knowledge is neither lost nor gained*}
   (P *knows* b *at* x) $=$ (P *knows* b *at* $(x;e)$). □

Proof. We prove only (1). Consider any z such that $(x;e)[P]z$. We will show b *at* z and hence it follows that P *knows* b *at* $(x;e)$.

Since $z[\bar{P}]\,z$, we have $(x;e)[P\bar{P}]\,z$.

From theorem 3, since e is a receive, $x[P\bar{P}]\,z$. Since b is local to $\bar{P}$,

P *knows* $b = P$ *knows* $\bar{P}$ *knows* b.

From theorem 4,

(P *knows* $\bar{P}$ *knows* b *at* x, $x[P\bar{P}]\,z$) *implies*
($\bar{P}$ *knows* b *at* z)

($\bar{P}$ *knows* b *at* z) *implies* (b *at* z).

This completes the proof. □

Corollary. (b *is local to* $\bar{P}$, $\sim P$ *knows* b *at* x, P *knows* b *at* y, $x \leqq y$) *implies* (P *receives a message in* (x, y)). □

Corollary. (b *is local to* $\bar{P}$, P *knows* b *at* x, $\sim P$ *knows* b *at* y, $x \leqq y$) *implies* (P *sends a message in* (x, y)). □

Theorem 5. *(How knowledge is gained) Let* x, y *be computations where* $x \leqq y$, $\sim(P_n$ *knows* $b)$ *at* x *and* (P_1 *knows* ... P_n *knows* b) *at* y, *for arbitrary process sets* $P_1 \ldots P_n$, $n \geqq 1$. *Then there is a process chain* $\langle P_n \ldots P_1 \rangle$ *in* (x, y). *Furthermore, if* b *is local to* $\bar{P}_n$ *then* P_n *has a receive event in* (x, y) *such that* b *at* z *holds for every prefix* z *of* y *which includes the corresponding send event.* □

Theorem 6. *(How knowledge is lost) Let* x, y *be computations where* $x \leqq y$, P_1 *knows* ... P_n *knows* b *at* x *and* $\sim P_n$ *knows* b *at* y, *for arbitrary process sets* $P_1 \ldots P_n$, $n \geqq 1$. *Then there is a process chain* $\langle P_1 \ldots P_n \rangle$ *in* (x, y). *Furthermore, if* b *is local to* $\bar{P}_n$ *then* P_n *has a send event in* (x, y). □

Observe that the statements of the two theorems are not entirely symmetric for receive and send events. The reason is that every computation including a receive must also include the corresponding send, but not conversely.

5 Applications of the results

We discuss a few applications of the theory developed so far in the paper.

5.1 When is a process unsure about a predicate?

We show that it is impossible for processes P to track the change in value of a local predicate of $\bar{P}$, at all times; P must be unsure about the value of this predicate while it is undergoing change.

Lemma 6. *(Interval of uncertainty:) Let* b *be a predicate local to* $\bar{P}$. *Let,* b *at* $x \neq b$ *at* $(x;e)$ *for some computation* $(x;e)$. *Then* P *unsure* b *at* x *and* P *unsure* b *at* $(x;e)$. □

Proof. Since b is local to $\bar{P}$ and its value changes as a result of event e, e is not on P. Therefore, $x[P](x;e)$ and hence P *knows* b *at* $x = P$ *knows* b *at* $(x;e)$. Since b *at* $x \neq b$ *at* $(x;e)$, both P *knows* b *at* x and P *knows* b *at* $(x;e)$ are *false*. Analogously, P *knows* $\sim b$ *at* x and P *knows*

$\sim b$ *at* $(x;e)$ are both *false*. This completes the proof. □

What does this lemma imply about the event e on $\bar{P}$ which changes the value of local predicate b of $\bar{P}$? It follows that P must be unsure about b for event e to occur. Furthermore, we show that if e is internal or send then a necessary condition for occurrence of e is that $\bar{P}$ *knows* P *unsure* b before application of e.

Theorem 7. *Let b be local to $\bar{P}$. For a computation $(x;e)$, where*

b at $x \neq b$ at $(x;e)$

($\bar{P}$ knows P unsure b) at x, if e is an internal or send event on $\bar{P}$,
($\bar{P}$ knows P unsure b) at $(x;e)$, if e is internal or receive on $\bar{P}$ □

Proof. Consider any y for which $x[\bar{P}]y$. From the principle of computation extension, $(y;e)$ is also a computation; hence $(x;e)[\bar{P}](y;e)$.

b is local to $\bar{P}$, hence: b *at* $x = b$ *at* y
and, b *at* $(x;e) = b$ *at* $(y;e)$.

From, b *at* $x \neq b$ *at* $(x;e)$ it follows that : b *at* $y \neq b$ *at* $(y;e)$.

Hence, from lemma (6), P *unsure* b *at* y.

From the definition of knowledge, $\bar{P}$ *knows* P *unsure* b *at* x. The other part is similarly proven. □

Corollary. *Let b be local to $\bar{P}$. For a computation $(x;e)$, where e is an internal event on $\bar{P}$, if:*

b at $x \neq b$ at $(x;e)$

then for any $y, x \leq y$, where $\bar{P}$ has no send event in (x,y):

$\bar{P}$ knows P unsure b at y. □

Proof. From Theorem 7, $\bar{P}$ *knows* P *unsure* b *at* x. Since $\bar{P}$ sends no message in (x,y), from Lemma 4, $\bar{P}$ can lose no knowledge and hence, $\bar{P}$ *knows* P *unsure* b *at* y. □

5.2 *Detection of process failure is impossible*

Traditional techniques for process failure detection based on time-outs assume certain execution speeds for processes and maximum delays for message transfer. It is generally accepted that detection of failure is impossible without using time-outs, a fact that we prove formally.

We model failure of $\bar{P}$ as follows. Let f be a local predicate of $\bar{P}$ denoting that $\bar{P}$ has failed. We assume that (1) f is initially *false*, and (2) $\bar{P}$ may fail at any time, i.e. for every x for which $\sim f(x)$, there is an internal event e on $\bar{P}$ such that $f(x;e)$ and (3) $\bar{P}$ sends no message as long as f holds. Under these constraints, we show that P is always *unsure* of failure of $\bar{P}$. In fact, we show that $\bar{P}$ *knows* P *unsure* f at all computations y. Note that we do not require failure to persist, i.e. it is entirely possible to have $x \leq y, f(x)$ and $\sim f(y)$.

Theorem 8. *$\bar{P}$ knows P unsure f at y, for all y,* □

Proof. If $\sim f(y)$, there is an internal event e on $\bar{P}$ such that $f(y;e)$. From Theorem 7, $\bar{P}$ *knows* P *unsure* f *at* y. If $f(y)$, then from the fact that f is *false* initially, there is some $(x;e)$, $(x;e) \leq y$, such that, $\sim f(x)$ and $f(x;e)$. Without loss in generality, we may assume that $\bar{P}$ stays failed after $(x;e)$ until y. Since e is an internal event and $\bar{P}$ stays failed after $(x;e)$, there is no send event on $\bar{P}$ in (x,y). Hence, from corollary to Theorem 7, $\bar{P}$ *knows* P *unsure* f *at* y. □

5.3 *Mutual exclusion*

Consider a system of processes in which every process p has a local predicate cs_p and for every pair of processes p, q and every computation x, $\sim(cs_p$ *and* $cs_q)$ *at* x. Intuitively, cs_p denotes that p is in its critical section and the restriction that no two processes can simultaneously be in their critical sections, is captured by the last requirement. We show that in every computation of a solution to the mutual exclusion problem (in our model), there is a process chain $\langle p_1 \ldots p_n \rangle$, where p_i is the ith process to enter its critical section.

Theorem 9. *For any $x, y, x \leq y$, cs_p at x and cs_q at y implies that there is a process chain $\langle pq \rangle$ in (x,y).* □

Proof. Observe that cs_p *implies* $\sim cs_q$, and $\sim cs_q$ *implies* (q *knows* $\sim cs_q$). Also, cs_q *implies* ($\sim q$ *knows* $\sim cs_q$). Hence, (cs_p *at* x) *implies* (p *knows* q *knows* $\sim cs_q$ *at* x) and (cs_q *at* y) *implies* ($\sim q$ *knows* $\sim cs_q$ *at* y). The result follows from theorem (6). □

We can show, based on the observation given below, that a solution to the distributed dining philosophers problem appearing in [1] requires no more than twice the number of messages in an optimal scheme. In the distributed dining philosophers problem, philosophers are placed at vertices of an undirected graph and one fork is placed on each edge. A philosopher requires forks on all incident edges to eat and hence neighboring philosophers cannot eat simultaneously.

Observation. For neighboring philosophers p, q, there is a process chain $\langle pq \rangle$ in (x, y) where p eats at x, q eats at y and $x \leq y$. Hence at least one message must be sent by p to q between an eating session by p and a subsequent eating session by q. The solution in [1] employs two messages between an eating session by p and a subsequent eating session by q.

5.4 *Complexity of termination detection*

We show that any algorithm which detects termination of an underlying computation requires at least as many overhead messages, in general, for detection as there are messages in the underlying computation. We prove our result by considering a specific underlying computation.

Consider a system of two processes A, B in which messages may be sent from A to B and from B to A. Each process is initially in a *tossing* state. Each process in *tossing* state decides nondeterministically (by a coin toss, for instance) to enter either a *receiving* or a *sending* state. A process in the *receiving* state waits until it receives a message and then returns to the tossing state. A process in the sending state sends a message and then returns to the tossing state. If both processes are in the receiving state and every message sent has been received, then both processes will remain waiting forever. The goal of the termination detection algorithm is to detect such a situation.

In the sequel, we use *underlying computation* to mean the computation associated with coin tossing, sending and receiving of messages as described above. The termination detection algorithm superimposes an *overhead computation* on the underlying computation at each process; we use *computation* to mean the underlying computation and overhead computation together. *Overhead messages* and *underlying messages* belong to the corresponding computations.

The overhead computation at a process can observe the state of the underlying computation, but cannot affect it. The overhead computation may have its own associated states and it may send messages (to the overhead computation at the other process) even when the underlying computation is waiting to receive. However, a message is received only when the underlying computation is waiting to receive. We require that whenever the termination detection algorithm reports termination, the underlying computation has terminated (both processes are in receiving state and there is no underlying message in channels); furthermore, for every computation x in which the underlying computation has terminated, there is a computation $y, x \leq y$, in which termination is reported by the overhead computation at one of the processes.

We show that for any $k, k \geq 0$, there is a computation in which k underlying messages are sent and received and at least k overhead messages are sent. The plan of the proof is as follows. We first show that in order for termination to be detected, an overhead message is sent by some process, without its first receiving a message, after the underlying computation terminates; this fact is proven directly from the theorem of knowledge gain, because detecting termination amounts to gaining knowledge.

Next, we show that a process is sometimes required to send an overhead message even when the underlying computation has not terminated, because the computation may be isomorphic (with respect to this process) to a computation in which the underlying computation has terminated. Using these two results, we construct a computation, of the required type, for any $k, k \geq 0$.

Theorem 10. *For any $k, k \geq 0$, there is a computation in which k underlying messages are sent and received and at least k overhead messages are sent.* □

Proofs. We will prove a slightly stronger result, $I(k)$, for any $k, k \geq 0$, where $I(k)$ is: there is a computation in which k underlying messages are sent and received, at least k overhead messages are sent and both processes are in tossing state at the end of the computation. Proof is by induction on k.

For $k = 0$: $I(0)$ holds for the *null* computation, from the initial condition.

Let x be a computation for which $I(k)$ holds for some $k, k \geq 0$. We show a computation z in which $I(k+1)$ holds.

Let $tr_A(tr_B)$ denote an internal event at $A(B)$ whereby the process transits from tossing state to receiving state; similarly, let $ts_A(ts_B)$ denote the transition from the tossing to sending state.

Consider the computation $x' = (x; tr_A; tr_B)$. Since no underlying message is in transit in x and both processes are waiting to receive in x', x' has a terminated underlying computation.

For each process, "process is in receiving state" is a local predicate of the process. This predicate value, for each process, is *false* at x. If a process (say B) detects termination at some $y, x' \leqq y$, then B *knows* A is in receiving state *at* y. Therefore, B gains knowledge about A and, applying the knowledge gain theorem (theorem 5), there is a process chain $\langle A, B\rangle$ in (x', y). Therefore, in general, either there is a process chain $\langle A\,B\rangle$ or a process chain $\langle B\,A\rangle$ in (x, y). Let y' be such that $x \leqq y' < y$, (x, y') contains no process chain $\langle A\,B\rangle$ or $\langle B\,A\rangle$ and (x, y') contains a message send (which must be an overhead message) by some process, say A.

Let $w = (x; tr_A; ts_B; B$ sends underlying message). Since there is no process chain $\langle A\,B\rangle$ or $\langle B\,A\rangle$ in (x, y') or (x, w), we can apply the fusion theorem (theorem 2) to y' and w to obtain a computation w', where $x \leqq w'$, $w'[B]w$ and $w'[A]y'$. In computation (x, w'), B has sent an underlying message and A has sent an overhead message before receiving the underlying message. To complete the proof, we note that there is an extension z of w' in which A receives the underlying message sent to it by B. Computation z satisfies $I(k+1)$. □

6 Discussion

We have shown that isomorphism between system computations with respect to a process is a useful concept in reasoning about distributed systems. Isomorphism forms the basis for defining and deriving properties about knowledge. "Scenarios" have been used [7] to show impossibility of solving certain problems; in our context, a scenario is a computation, and isomorphism is the formal treatment of equivalence between scenarios. Theorems on knowledge transfer provide lower bounds on numbers of messages required to solve certain problems. We have used isomorphism as the basis of fusion theorem and related isomorphism to semantics of send, receive and internal events.

In this paper, we have not defined processes in terms of their states. The notion of isomorphism between computations could be defined in terms of process states as follows: two computations x and y are *state-isomorphic* with respect to a process p means the state of p after x is the same as its state after y. Observe that x and y are *isomorphic* with respect to a process implies they are *state-isomorphic* with respect to that process. With knowledge defined in terms of state-isomorphism, a process may lose knowledge by an internal event, that is, by merely by changing its state. However, knowledge can be gained only be receiving messages. In other words, processes may "forget" on their own but cannot learn without receiving information. The theorem of knowledge transfer applies even with knowledge defined in terms of *state-isomorphism*. This is an area worth pursuing, as it may provide insight into designs of processes.

Our model does not have the notion of time. If there is a global clock common to all processes then processes may learn or forget merely by the passage of time. For instance, in time-division multiplexing, the mutual exclusion problem is solved by letting the i-th process be in its critical section during the i-th slot in the time cycle. In this case, a computation is a tuple consisting of the "current" time and a sequence of timed-events where each timed-event is a pair (time, event). The concept of isomorphism remains valid, though the knowledge transfer theorems no longer hold, because knowledge can be gained and lost merely by the passage of time.

It is tempting to define *belief* in terms of isomorphism as follows: process p *believes* b at x means b holds for most (in measure-theoretic terms) computations isomorphic to x with respect to p. Unfortunately, there do not appear to be clean results on the gain/loss of belief or belief transfer.

In this paper, when we say a process knows b, we allow b to be an arbitrary predicate: b may be temporal, for instance of the form: *eventually* b'. For example, in a commit protocol a process committing itself to a value v knows that all correct processes will eventually commit to v. Results about knowledge transfer-gain or loss-still hold.

Acknowledgement. We are indebted to Shmuel Katz, Joe Halpern, E.W. Dijkstra and Bengt Jonsson for their comments. Particular thanks go to Ernie Cohen for a careful reading of the manuscript and insightful comments.

References

1. Chandy KM, Misra J (1984) Drinking philosophers problem. TOPLAS, October 1984
2. Fischer MJ, Lynch N, Paterson M (1985) Impossibility of distributed consensus with one faulty process. J ACM, April 1985
3. Halpern JY, Moses Y (1984) Knowledge and common knowledge in a distributed environment. ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Vancouver, Canada, August 1984
4. Hintikka J (1962) Knowledge and belief. Cornell University Press
5. Lamport L (1978) Time, Clocks and the orderings of events in a distributed system. Communications of the ACM 21: 558-564
6. Lehmann D (1984) Knowledge, common knowledge, and related puzzles. ACM SIGACT-SIGOPS Symposium of Principles of Distributed Computing, Vancouver, Canada, August 1984
7. Lynch N & Fischer M (1982) A lower bound for the time to assure interactive consistency. Information Proc Letters 14, 4, June 1982

ference Record
of the

nual ACM Sy
on
ogramming

nted at the Symp

Correction to the Paper:

A Really Abstract Concurrent Model and its Temporal Logic

by: Barringer, Kuiper and Pnueli

Not all the positive operators of the Real Temporal logic are continuous. All of them are monotonic. We distinguish two types of continuity. An operator $\varphi(X)$ is defined to be $\vee$-continuous if it satisfies

$$\varphi(\bigvee_{i<\omega} p_i) = \bigvee_{i<\omega} \varphi(p_i)$$

It is defined to be $\wedge$-continuous if it satisfies

$$\varphi(\bigwedge_{i<\omega} p_i) = \bigwedge_{i<\omega} \varphi(p_i)$$

The following operators are $\vee$-continuous:

$$\vee, \wedge, \diamondplus, \diamondminus, \lambda X.(p\hat{\mathcal{U}}X), \lambda X.(p\hat{\mathcal{S}}X)$$

The following operators are $\wedge$-continuous:

$$\vee, \wedge, \boxplus, \boxminus, \lambda X.(X\hat{\mathcal{U}}q), \lambda X.(X\hat{\mathcal{S}}q)$$

A general equation: $X \equiv \varphi(X)$ where φ is a *monotonic* operator has both a minimal and a maximal fixpoint solutions, denoted by $\mu X.\varphi$ and $\nu X.\varphi$ respectively. These solutions can be obtained by limits of approximations which for a general monotonic operator must be carried to an ordinal order. They can be defined by:

$$\varphi^0_\vee(X) = X,$$

For a non-limit ordinal (or finite index)

$$\varphi^{\alpha+1}_\vee(X) = \varphi(\varphi^\alpha(X))$$

For a limit ordinal β

$$\varphi^\beta_\vee(X) = \bigvee_{\alpha<\beta} \varphi^\alpha_\vee(X)$$

For every monotonic φ there exists an ordinal α such that

$$\mu X.\varphi = \varphi_{\vee}^{\alpha}(F)$$

If φ is also $\vee$-continuous then:

$$\mu X.\varphi = \varphi_{\vee}^{\omega}(F) = \bigvee_{i<\omega} \varphi^{i}(F)$$

Similarly, for approximating the maximal fixpoint we define:

$$\begin{array}{ll} \varphi_{\wedge}^{0}(X) = X & \\ \varphi_{\wedge}^{\alpha+1}(X) = \varphi(\varphi_{\wedge}^{\alpha}(X)) & \text{For a non-limit ordinal } \alpha + 1 \\ \varphi_{\wedge}^{\beta}(X) = \bigwedge_{\alpha<\beta} \varphi_{\wedge}^{\alpha}(X) & \text{For a limit ordinal } \beta \end{array}$$

For every monotonic φ, there exists an ordinal α such that

$$\nu X.\varphi = \varphi_{\wedge}^{\alpha}(T)$$

If φ is also $\wedge$-continuous then:

$$\nu X.\varphi = \varphi_{\wedge}^{\omega}(T) = \bigwedge_{i<\omega} \varphi^{i}(T)$$

A Really Abstract Concurrent Model and its Temporal Logic

Howard Barringer(1)
Ruurd Kuiper(1)
Amir Pnueli(2)

Extended Abstract
July 1985

(1) University of Manchester, Manchester, England
(2) Weizmann Institute of Science, Rehovot, Israel

Abstract. In this paper we advance the radical notion that a computational model based on the *reals* provides a more abstract description of concurrent and reactive systems, than the conventional *integers* based behavioral model of execution *sequences*. The real model is studied in the setting of temporal logic, and we illustrate its advantages by providing a *fully abstract* temporal semantics for a simple concurrent language, and an example of verification of a concurrent program within the real temporal logic defined here. It is shown that, by imposing the crucial condition of *finite variability*, we achieve a balanced formalism that is insensitive to *finite* stuttering, but can recognize *infinite* stuttering, a distinction which is essential for obtaining a fully abstract semantics of non-terminating processes. Among other advantages, going into real-based semantics obviates the need for the controversial representation of concurrency by interleaving, and most of the associated fairness constraints.

Introduction

Temporal logic is, by now, a widely accepted formal tool for the specification and verification of concurrent and reactive systems (see [MP1], [La1], [OL], [HO], [SMS], [CE], [CM] and many others). The underlying time structure upon which those systems are based is *discrete*, and, in the linear temporal logic case, is isomorphic to the non-negative integers and models the execution sequences that the specified program generates.

The research was supported in part by SERC grant GR/C/05760.

Part of the research of the third author was supported by ONR grant N00014-85-K-0057 while visiting the University of Texas at Austin.

An important step in the construction and justification of temporal proof systems is the definition of *temporal semantics*, which constructs for a given program P a characteristic formula ϕ_P, sometimes denoted by $[\![P]\!]$, such that ϕ_P is true precisely over all the admissible executions of P. Such definitions have been given for global systems in [Pn1], [MP2], and in a more syntax directed style, suitable to compositional proof systems in [BKP1], [BKP2].

When comparing the temporal semantics of concurrent programs with other semantic definitions we find that they are deficient in one respect. Namely, they do not achieve *full abstractness*. Full abstractness ([M1]) is a most important criterion which requires that the semantics level of detail should match the desired level of abstractness. In particular it requires that any two programs that we wish to consider equivalent, should be assigned identical semantics. For sequential programs we can easily say that it was the strive towards full abstractness that led from the overly detailed operational semantics into the much more satisfactory denotational domain-based semantics.

Consider the following two program segments that represent modules in a concurrent program:

$$P_1 :: \; x := 1;\; x := x;\; x := 2 \quad , \text{and}$$
$$P_2 :: \; x := 1;\; x := x;\; x := x;\; x := 2$$

They differ by the number of dummy $x := x$ assignments separating the two externally observable instructions $x := 1$ and $x := 2$. At the qualitative level that we want to analyze such concurrent programs, these two program segments should be considered equivalent.

Let us examine whether their temporal semantics are indeed identical. We consider first the logic $\mathcal{L}^{\oplus} = \mathcal{L}(\bigcirc, \mathcal{U})$ presented in [MP1] and other related works. This logic uses the basic operators $\bigcirc$ (next time) and $\mathcal{U}$ (until), over an integer-like execution sequence.

Without giving the precise temporal semantics of P_1 and P_2 we can still explain how they differ. The temporal semantics of P_2 (in the $\mathcal{L}^{\oplus}$ logic) requires that in any computation sequence of P_2, the $x = 1$ and $x = 2$ are separated by *at least* 3 computation steps (or two intermediate states). In P_1 the lower bound is only two computation steps. Consequently the $L^{\oplus}$ semantics distinguishes between P_1 and P_2, and hence is not fully abstract.

Lamport perceived this lack of abstractness in the $L^{\oplus}$ logic and attributed the problem to the next-time operator. Consequently, the temporal logic that he works with ([Lal], [OL]) is $L^{+} = L(\mathcal{U})$, which uses only the until operator (or an appropriate equivalent). He also formulated the requirement that, in order to be abstract, the logic must be insensitive to *stuttering*, which he defined as finite consecutive duplication of some states. Indeed any execution sequence of P_2 may be obtained from some execution of P_1 by duplication of some states, and the semantics that would be assigned to P_1 and P_2 in the L^{+} logic are identical:

$$[\![P_1]\!] = [\![P_2]\!] = \Diamond[(x = 1)\mathcal{U}^{+}(x = 2)]$$

where we use the defined operator $p\mathcal{U}^{+}q = (p \wedge p\mathcal{U}q)$.

Unfortunately, while this next-less logic provides an abstract semantics for finite processes, i.e., having bounded executions, it raises new problems when we go to infinite processes. We may interpret the view represented by Lamport's approach by saying that there is no *absolute* time scale against which executions are measured. Time advances only when there is a (state) change. Clearly, such a view would naturally ignore any *finite* periods of no change. However, by the same token, it would also ignore (or collapse) *infinite* periods of no change, which is unacceptable.

Consider the following recursive procedure

$$P \Leftarrow [x := x;\ P]$$

where it is assumed that the process P *owns* the variable x, in the sense that P is the only process which may modify x.

The common association of semantics to such a procedure is to form a fixpoint equation for a temporal predicate, where the right hand side of the equation is obtained from the semantics of the procedure's body. We therefore look for the *maximal* solution to the equation:

$$X \equiv \exists u.[(x = u)\mathcal{U}^{+}((x = u) \wedge X)]$$

It is not difficult to see that the maximal solution is $X = T$, i.e., all possible behaviors, in particular those that arbitrarily modify x. This can be explained by the fact that the procedure P produces infinite stuttering which the L^{+} semantics consumes in zero time, and leaves the rest of the execution unrestricted.

If in comparison we consider the semantics assigned to this process by the $L^{\oplus}$ logic, we replace $\mathcal{U}^{+}$ by $\mathcal{U}^{\oplus}$, defined as $r\mathcal{U}^{\oplus}q \equiv r \wedge \bigcirc(r\mathcal{U}q)$. Then the solution is

$$X = \exists u.\ \Box(x = u)$$

i.e., x is continuously preserved, which is what we intuitively expect.

Thus we find that $L^{\oplus}$ is unsatisfactory because it is sensitive to finite stuttering, while L^{+} is unsatisfactory because it is insensitive to infinite stuttering.

These difficulties are not specific to temporal semantics. To the best of our knowledge, no adequate compositional semantics of concurrent programs which satisfies all of the four following requirements, has yet been proposed.

1. Allows nondeterminism in the processes.
2. Treats fair parallelism.
3. Is fully abstract, in particular is insensitive to finite stuttering.
4. Properly treats divergent processes, in particular infinitely stuttering processes.

Most of the works that did propose semantics of concurrent programs are usually deficient in points 2 or 4 or both. Point 4 is of course highly subjective, and we have our own interpretation of what the "proper" treatment of nonterminating processes should be. By this interpretation nontermination should not be considered catastrophic, and a silently divergent (infinitely stuttering) process should have no effect on any process running in parallel, except when termination of the full system is considered, which we ignore in this treatment. Thus if we define the silently divergent process:

$$1 = [P \textit{ where } P \Leftarrow [\ \textit{skip}\ ; P]]$$

we would like to have

$$(1 \parallel Q) \approx Q$$

for any process Q.

Usually, in works such as [HM], [dBMO], [Br], the silently divergent process 1 is treated either as catastrophic (the Smyth view) or as *chaos* (completely unspecified process) which leads to equivalences of the form

$$(1 \parallel Q) \approx 1.$$

In our previous work ([MP2], [BKP1], [BKP2]) using $L^{\oplus}$ we usually achieved requirements 1, 2, and 4 but had to give up on 3. In this paper we suggest that linear temporal logic with the time structure of the (non negative) *real numbers* provides a more abstract logic than that of the non negative integers, and succeeds in meeting all the four criteria above.

Temporal Logic of the Reals (TLR)

Let $V = L \cup G$ be a set of variables which is partitioned into $L = \{y_1, \ldots\}$ the *local* variables, and $G = \{u_1, \ldots\}$ the *global* variables. For simplicity we assume that some of the variables range over a *data* domain, and the others, which we call *propositions*, range over the boolean domain $\{F, T\}$.

A *model* over V is an assignment α that assigns to each variable $v \in V$ and each non-negative real number $t \geq 0$, a value $\alpha(v, t)$ from the appropriate domain. The assignment α is required to satisfy:

a) Uniformity of global interpretation –
For each *global* variable $u \in G$, $\alpha(u, t)$ is independent of t.

b) Finite variability –
for each local variable $y \in L$ there exists a denumerable sequence:

$$0 = t_0 < t_1 < t_2 < \ldots \quad \text{with } t_n \to \infty$$

such that the value of $\alpha(y, t)$ is uniform within each open interval (t_i, t_{i+1}), i.e., for every t, t', if $t_i < t < t' < t_{i+1}$ then $\alpha(y, t) = \alpha(y, t')$.

Condition b) guarantees that there could be no infinite variability within a finite interval, and that the interpretation of each variable can be decomposed into countably many open intervals of constant value. Note that we do not restrict the values at the break-points t_i, which could be different from the values of their left or right neighbor intervals.

The temporal logic we consider is based on the operators $\hat{U}$ (until) and $\hat{S}$ (since) ([LPZ]).

We define the value of terms and state formulae at a nonnegative real instant t of a model α by evaluating them pointwise, i.e., using $\alpha(v_i, t)$ whenever the value of v_i is needed. For a state formula φ, we denote by $\varphi(\alpha, t)$ the value obtained by such pointwise evaluation at point t. *Then we define satisfiability as follows:*

$(\alpha, t) \models \phi$	*iff*	$\phi(\alpha, t) = T$ where ϕ is a state formula
$(\alpha, t) \models \neg\phi$	*iff*	$(\alpha, t) \not\models \phi$
$(\alpha, t) \models (\phi_1 \vee \phi_2)$	*iff*	$(\alpha, t) \models \phi_1$ or $(\alpha, t) \models \phi_2$
$(\alpha, t) \models \phi \hat{U} \psi$	*iff*	$\exists t'', t < t''$, such that $(\alpha, t'') \models \psi$ and for every $t', t < t' < t''$, $(\alpha, t') \models \phi$
$(\alpha, t) \models \phi \hat{S} \psi$	*iff*	$\exists t'', 0 \leq t'' < t$, such that $(\alpha, t'') \models \psi$ and for every $t', t'' < t' < t$, $(\alpha, t') \models \phi$

Note that differently from the integer-based TL, the basic *until* operator $\varphi \hat{U} \psi$ is strict and guarantees a non empty φ interval. We may also define some derived operators:

$$\begin{array}{ll} \varphi \wedge \psi = \neg(\varphi \vee \psi) & \varphi \to \psi = (\neg\varphi \vee \psi) \\ \blacklozenge \varphi = T \hat{U} \varphi & \ominus \varphi = T \hat{S} \varphi \\ \boxplus \varphi = \neg \blacklozenge \neg\varphi & \boxminus \varphi = \neg \ominus \neg\varphi \\ \varphi \mathcal{U}^{+} \psi = \varphi \wedge \varphi \hat{U} \psi & \end{array}$$

The derived temporal operators $\blacklozenge$, $\boxplus$ have similar meaning to that of their integer-based counterparts, except that in real temporal logic they are strict, meaning that the present (point t) is not considered as a part of the future.

Two additional logical operators that are needed are *quantification* and *fixpoint*.

The semantics of the existential quantifier is given by:

$(\alpha, t) \models \exists v.\varphi$ **iff** there exists a model α' differing from α by at most the assignment given to v, such that $(\alpha', t) \models \varphi$

Note that we allow quantification over both global and local variables, in contrast to [MP1] where quantification is allowed only over global variables. When quantifying over a local variable y, the requirement that α' be a model according to the definition given above implies that v satisfies the finite variability condition.

Universal quantification may be introduced as a derived operation:

$$\forall v.\varphi = \neg\exists v.(\neg\varphi)$$

In order to define the fixpoint operator it helps to slightly shift our view of the semantics of temporal formulae and define for each formula φ and a non-negative real number $t \geq 0$, their *extent* (validity-set) given by:

$$E(\varphi, t) = \{\alpha \mid (\alpha, t) \models \varphi\}$$

This definition associates with each formula φ and time instant $t \geq 0$, a set of all the possible models that satisfy φ at t. This leads to a view by which each formula φ defines a function E_φ from R^+ (the non-negative reals) to $\mathcal{M}$, the set of all models (over V). Let $\mathcal{D} = [R^+ \to \mathcal{M}]$ denote the set of *all* functions from R^+ to $\mathcal{M}$. It is not difficult to see that it is a complete lattice, actually a complete boolean algebra. The ordering on $\mathcal{D}$ is closely connected to implication between formulas. Thus $\varphi \sqsubseteq \psi$ (when interpreted as elements of $\mathcal{D}$) *iff* $\varphi \to \psi$ is valid. Consequently the minimal element of $\mathcal{D}$ is $F = \lambda t.\phi$ and the maximal element of $\mathcal{D}$ is $T = \lambda t.\mathcal{M}$.

The logical operators may now be viewed as functions from $\mathcal{D}$ to $\mathcal{D}$. Thus for ever two elements $e_1, e_2 \in \mathcal{D}$, we may express the operators of disjunction and *until* by:

$$e_1 \vee e_2 = e_1 \sqcup e_2 = \lambda t.\{\alpha \mid \alpha \in e_1(t) \quad \text{or} \quad \alpha \in e_2(t)\}$$

$$e_1 \hat{U} e_2 = \lambda t.\{\alpha \mid \exists t''[t < t'', \alpha \in e_2(t'') \text{ and } \forall t', t < t' < t'', \alpha \in e_1(t')]\}$$

We can show that all the operators defined above excluding $\neg$ are continuous, while $\neg$ is anti-continuous over $\mathcal{D}$. Consequently, we consider equations of the form:

$$X \equiv \varphi(X)$$

where X is a local proposition variable, and φ is a temporal formula in which all instances of X are positive, i.e., encompassed by an *even* number of negations. In such a case this equation is known to have both a *minimal* and a *maximal* solution. We denote them respectively by $\mu X.\varphi$ and $\nu X.\varphi$. The usual property of extremal fixpoints being obtained by limits of finite approximations, can be expressed in our ~~case by~~:

$$\mu X.\varphi \equiv \bigvee_{i=0}^{\infty} \varphi^i(F)$$

and

$$\nu X.\varphi \equiv \bigwedge_{i=0}^{\infty} \varphi^i(T)$$

where for $\varphi = \varphi(X)$ we define $\varphi^0(X) = X$ and $\varphi^{i+1}(X) = \varphi(\varphi^i(X))$.

As a simple example consider the equation

$$X \equiv \blacklozenge(p \wedge X)$$

Its maximal fixpoint can be obtained by approximations. Denoting $\varphi(X) = \blacklozenge(p \wedge X)$, we can show that $\varphi^i(T)$ holds at t *iff* there are i distinct time instants $t < t_1 < \ldots < t_i$ such that p holds at each of the $t_1, \ldots, t_i$. Consequently $\nu X.\varphi$ holds at t *iff* there are infinitely many points ahead of t at which p is true. In real temporal logic this leads to:

$$\nu X.\blacklozenge(p \wedge X) \equiv (\boxplus \blacklozenge p \vee \blacklozenge(p\hat{U}T))$$

On the other hand, the minimal fixpoint of this equation is F. This is due to the fact that F satisfies the equation and is also the minimal element of D. We thus have:

$$\mu X.\blacklozenge(p \wedge X) \equiv F$$

An important observation is that all the operators introduced respect the finite variability restriction. This means that the finite variability restriction holds not only for the propositions and variables, but also for any temporal formula defined over them.

Axiomatic Characterisation of the Real Temporal Logic

Whenever a logic is introduced and recommended as a tool for formal reasoning about programs, an essential part of this recommendation should be a deductive system that supports sound reasoning within the logic itself. Since the full logic is clearly not finitely axiomatizable, we will introduce the deductive system we propose in steps, indicating the step at which we lose completeness and decidability.

The Propositional Fragment

The propositional fragment is obtained by requiring that all the variables in V are propositions, i.e., range over $\{F, T\}$. In this case *global* quantification can be eliminated. This is because for a global proposition u:

$$\exists u.\varphi(u) \equiv \varphi(T) \vee \varphi(F).$$

Consider first the language without (local) quantification or fixpoint operators. We propose the following axiomatization:

F0. All substitution instances of propositional tautologies.

F1. $\boxplus(\varphi \to \psi) \to \{[\varphi\hat{U}\theta \to \psi\hat{U}\theta] \wedge [\theta\hat{U}\varphi \to \theta\hat{U}\psi]\}$

F2. $\varphi \wedge \theta\hat{U}\psi \to \theta\hat{U}[\psi \wedge \theta\hat{S}\varphi]$

F3. $\varphi\hat{U}\psi \equiv [\varphi \wedge \varphi\hat{U}\psi]\hat{U}\psi$

F4. $\varphi\hat{U}\psi \equiv \varphi\hat{U}[\varphi \wedge \varphi\hat{U}\psi]$

F5. $(\varphi\hat{U}\psi) \wedge \neg(\theta\hat{U}\psi) \to (\varphi \wedge \neg\theta)\hat{U}\psi$

F6. $\varphi\hat{U}\psi \wedge \theta\hat{U}\rho \to (\varphi \wedge \theta)\hat{U}(\psi \wedge \rho) \quad \vee \quad (\varphi \wedge \theta)\hat{U}(\psi \wedge \theta)$
$\qquad \vee \quad (\varphi \wedge \theta)\hat{U}(\varphi \wedge \rho)$

Six additional axioms P1–P6 are obtained as the mirror images of F1–F6, that is, by interchanging in each axiom $\boxplus$ with $\boxminus$ and $\hat{U}$ with $\hat{S}$.

Axioms F1 and P1 state that the $\hat{U}$ and $\hat{S}$ operators are monotonic in both arguments.

Axioms F2 and P2 specify the relation of reflection holding between past and future. Axioms F3, F4 and their past counterparts characterize the time structure as being *dense*, i.e., between every two instants there exists an additional instant distinct from both. To see this, consider a simpler version $\blacklozenge\varphi \to \blacklozenge\blacklozenge\varphi$ which also characterizes density. It certainly does not hold in integer-based TL, when we interpret $\blacklozenge\varphi$ as $\bigcirc\Diamond\varphi$. Axioms F6 and P6 state that the time structure is *linear*. Essentially it says that if both ψ and ρ are bound to happen, then they will either happen simultaneously or one will precede the other.

F7. $\boxplus\varphi \to \blacklozenge\varphi$

P7. $\boxminus F \vee \blacklozenge \boxminus F$

Axiom F7 states that the future is unbounded while P7 asymmetrically states that the past does have a definite starting point.

the proposed system includes the following inference rules:

R1. Modus Ponens: $\vdash \varphi, \vdash (\varphi \to \psi) \Rightarrow \vdash \psi$.

R2. Generalization: $\vdash \varphi \Rightarrow \vdash \boxplus\varphi, \vdash \boxminus\varphi$.

the system consisting of axioms F0–F7, P1–P7 and rules R1, R2 is taken almost verbatim from [Bu], where it is stated that it forms a sound and complete axiomatic system for the considered fragment of propositional TL over the *rational* half line $Q^+ = \{r \in Q \mid r \geq 0\}$.

In order to characterize the real half line $\mathbb{R}^+$ we usually add a *completeness* axiom. This axiom states that any (Dedekind) cut defined by a change of a proposition, say from T to F, identifies an instant *belonging* to the structure which marks the transition point. In our case, the requirement of finite variability already ensures that any change in value of a variable y must be associated with some node t_i that marks the transition point. Consequently completeness is superceded by the finite variability requirement represented by the axioms:

F8. $\varphi\hat{U}T \vee (\neg\varphi)\hat{U}T$

P8. $(\diamondminus T) \rightarrow \varphi\hat{S}T \vee (\neg\varphi)\hat{S}T$

These axioms state that for every formula φ and instant $t \geq 0$, there is always an open interval to the right of t ($\{t' \mid t < t' < t''\}$ for some t'', $t < t''$) in which the value of φ is uniform, and if $t > 0$, also an open interval to the left of t in which φ is uniform.

A consequence of the fact that finite variability implies completeness is that, relative to the language TLR, the class of models based on the reals is equivalent to the class of models based on the rationals. This means that a TLR formula is satisfiable by a real model iff it is satisfiable by a rational model. Consequently we may interpret the R of TLR as standing for either the Reals or the Rationals.

Consider next the introduction of the fixpoint operators to our system. Since the minimal and maximal fixpoint operators are interdefinable, we choose as basic the maximal fixpoint operator. It is controlled by the following axiom:

X1. $\nu X.\varphi(X) \equiv \varphi(\nu X.\varphi(X))$
i.e., the maximal solution to the equation $X \equiv \varphi(X)$ satisfies the equation.

A rule associated with the fixpoint operator is:

R3. $\vdash \theta \rightarrow \varphi(\theta) \;\Rightarrow\; \vdash \theta \rightarrow \nu X.\varphi(X)$

This rule states that $\nu X.\varphi(X)$ is the maximal solution to the equation $X \rightarrow \varphi(X)$, and hence every other solution, such as θ above, is smaller than $\nu X.\varphi(X)$.

The minimal fixpoint can be defined by:

$$\mu X.\varphi(X) = \nu X.\neg\varphi(\neg X)$$

The completeness of the system up to this point is discussed in [LP].

The most complex operators in the language are the quantifiers. Actually, the fixpoint operators can be defined by means of quantifiers. Introducing the abbreviation:

$$\Box\varphi = \boxminus\varphi \wedge \varphi \wedge \boxplus\varphi$$

we can express $\nu X.\varphi(X)$ by the following formula:

$$\exists q.[q \wedge \Box\epsilon(q) \wedge \forall p.\, \Box(\Box\epsilon(p) \rightarrow \Box(p \rightarrow q))]$$

where $\epsilon(r)$ is given by $r \equiv \varphi(r)$.

This formula explicitly states that q holds now, q satisfies the equation ϵ at all points, and any other p satisfying ϵ at all points is necessarily smaller or equal to q.

The axioms controlling the quantifiers are similar to those presented in [MP3]:

QF1. $\exists p.[\varphi\hat{U}\psi] \equiv \varphi\hat{U}(\exists p.\psi)$

where p is not free in φ.

The additional axiom QP1, is the past counterpart of QF1.

Q2. $\forall p.\varphi(p) \rightarrow \varphi(\theta)$

where θ is any formula free for p in φ.

And we also have the following rule:

R4. $\vdash \varphi \rightarrow \psi \;\Rightarrow\; \vdash \varphi \rightarrow \forall p.\psi$

where p is not free in φ.

For the proper definition of the semantics of programs we should be able to establish the existence of propositions that have an infinite variation.

For a formula φ, we define the following abbreviations:

$$\text{Rise}(\varphi) = [(\neg\varphi)\hat{S}T \vee (\neg\varphi)] \wedge [\varphi \vee \varphi\hat{U}T]$$

Rise(φ)is true at $t \geq 0$ *iff* t is a transition point at which φ changes from F to T.

$$\text{Fall}(\varphi) = \text{Rise}(\neg\varphi).$$
$$\text{Ch}(\varphi) = \text{Rise}(\varphi) \vee \text{Fall}(\varphi)$$

Thus Ch(φ) is true at $t \geq 0$ *iff* φ changes its value at t.

$$\text{Clock}(q) = [\boxplus \diamondminus q \wedge \Box(q \rightarrow (\neg q)\hat{U}T)]$$

A proposition is called a *clock* if it is true at infinitely many points, and whenever it is true it is immediately false at a right neighboring interval. This implies that q is true at countably many isolated points (never at an interval) and false elsewhere.

We add the following axiom:

C. $\exists q.\{\text{Clock}(q) \wedge \Box(\text{Ch}(\varphi) \rightarrow (\neg\text{Ch}(\varphi))\hat{U}(q \wedge \neg\text{Ch}(\varphi)))\}$

This axiom states that for any formula φ there exists a clock q that becomes true (ticks) at least once between every two consecutive changes in φ.

The questions of decidability and completeness of this axiomatic system for the propositional fragment of TLR will be discussed in [LP], hoping to establish positive answers for both.

A trivial extension of the propositional fragment, which is still decidable, is obtained by allowing a single fixed data domain D of finite cardinality.

The General Logic

As soon as we allow data domains of unbounded cardinality, the logic becomes highly undecidable and not finitely axiomatizable. In that case we have also to consider quantification over global variables. This quantification obeys axioms QF1, QP1, Q2 and rule R4 as well.

A formula φ is called *global* if it depends only on global variables and propositions. For global formulas φ we have the following axiom:

G. $\varphi \equiv \Box\varphi$

A Programming Language and Its Operational Semantics

We introduce a simple programming language of processes which communicate by shared variables. since we want to emphasize their continuous behavior rather than the result they yield on termination, we will not allow them to terminate.

Assuming that the syntax for terms and conditions is understood, the following recursive definition describes the syntax of processes:

Idle: *rest* is a process that performs no further action.

Call: *call P* represent a recursive call to a process P within its body.

Skip: If π is a process then so is *skip*; π

Assignment:

If y is a *data variable*, e a term and π a process, then $y := e; \pi$ is a process that first assigns e to y and then proceed to perform π.

Conditional:

If $\pi_1, \ldots \pi_k$ are processes and $c_1, \ldots, c_k$ are conditions, then $[\overset{k}{\underset{i=1}{\Box}} c_i \rightarrow \pi_i]$ is a process that nondeterministically chooses a direction i such that c_i is true and then proceeds to perform π_i, for some i, $i = 1, \ldots, k$.

Parallel:

If π_1, π_2 are two processes and $\overline{y^1}, \overline{y^2}$ two disjoint sets of data variables, then $[\mathit{own}\ \overline{y^1}; \pi_1 \parallel \mathit{own}\ \overline{y^2}; \pi_2]$ is a process that performs π_1 and π_2 in parallel. The *own* declarations partition the available variables into two sets associating with each process the set of variables it is allowed to modify.

Data Variables Declaration:

If π is a process then so is *new* $\bar{y}; \pi$, declaring a set of new variables $\bar{y}$ and then proceeding to perform π.

Process Declaration and Activation: If P is a process variable and B a process (body) then $[P \textit{ where } P \Leftarrow B]$ is a process that begins to perform B and recursively reactivate B whenever it meets a call to P. Note that our recursive processes do not admit parameters, and also never return from a call.

A *complete* process will have the general form *own* $\bar{y}; \pi$, where the preceding *own* declaration identifies the variables, not locally declared in π, which π may modify.

Given a complete process we define for each constituent subprocess ρ, the set mod(ρ) which is the set of variables that ρ actually modifies or declares owning.

This is defined by the following equations:

mod $(\mathit{rest}) = \emptyset$
mod $(\mathit{call}\ P) =$ mod(B) when the *call P* is
contained within a $P \Leftarrow B$ declaration.
mod $(\mathit{skip}; \pi) =$ mod(π)
mod $(y := e; \pi) =$ mod$(\pi) \cup \{y\}$
mod $([\overset{k}{\underset{i=1}{\Box}} c_i \rightarrow \pi_i]) = \bigcup_{i=1}^{k}$ mod(π_i)
mod $(\mathit{own}\ \bar{y}; \pi) =$ mod$(\pi) \cup \{\bar{y}\}$
mod $(\pi_1 \parallel \pi_2) =$ mod$(\pi_1) \cup$ mod(π_2)
mod $(\mathit{new}\ \bar{y}; \pi) =$ mod$(\pi) - \{\bar{y}\}$
mod $(P \textit{ where } P \Leftarrow B) =$ mod(B)

These equations are recursive, so we look for their *minimal* solution.

We may now define for each subprocess ρ the set owns(ρ), which is the set of variables that the context in which ρ occurs has declared as owned by ρ. The computation of these sets proceeds in a top-down fashion.

If $\rho = \mathit{skip}; \pi$ or $\rho = [y := e; \pi]$, then
owns$(\pi) =$ owns(ρ)
If $\rho = [\overset{k}{\underset{i=1}{\Box}} c_i \rightarrow \pi_i]$, then
owns$(\pi_i) =$ owns(ρ) for each $i = 1, \ldots, k$.
If $\rho = [\mathit{own}\ \bar{y}^1; \pi_1 \parallel \mathit{own}\ \bar{y}^2; \pi_2]$, then we require that
owns$(\rho) = \bar{y}^1 \cup \bar{y}^2$ and define
owns$(\pi_i) = \bar{y}^i$, for $i = 1, 2$
If $\rho = \mathit{new}\ \bar{y}; \pi$, then
owns$(\pi) =$ owns$(\rho) \cup \{\bar{y}\}$.
If $\rho = [P \textit{ where } P \Leftarrow B]$, then

$$\text{owns}(P) = \text{owns}(B) = \text{owns}(\rho) \cup \bigcup_{\mathit{call}P \in B} \text{owns}(\mathit{call}P)$$

This definition is again recursive ans we look for the minimal solution.

A complete process *own* $\bar{x}; \rho$ is well formed if:

a) No declaration of the form *new* $\bar{y}$ falls under the scope of another declaration for some variable in $\bar{y}$. Violations of this condition can always be corrected by renaming.

b) Every *call P* process occurs within the body of a declaration for P.

c) For every subprocess π in ρ (mod $(\pi)) \subseteq \mathit{owns}(\pi)$.

d) All the free variables in ρ are contained in $\bar{x}$.

We next define operational semantics for this language. We assume that each subprocess within the complete process *own* $\bar{x}; \rho$ is uniquely identifiable. We define a labelled transition relation representing the possible transformations that can be effected in one computation step. Assume a set of states S, each of which is a mapping from the currently declared variables to their values. A ***configuration*** is a pair $<\pi,\sigma>$ consisting of a process π and a state $\sigma \in S$.

For $\pi = [skip\ ; \rho]$ or $\pi = [own\ \bar{y}\ ; \rho]$,

$$<\pi,\sigma> \xrightarrow{\pi} <\rho,\sigma>$$

For $\pi = [y := e\ ;\ \rho]$,

$$<\pi,\sigma> \xrightarrow{\pi} <\rho, (\sigma; y{:}\,\sigma(e))>$$

where $(\sigma; y{:}\,\sigma(e))$ denotes the state obtained from σ by assigning the value of e evaluated at σ to y.

For $\pi = [\overset{k}{\underset{i=1}{\Box}} c_i \rightarrow \rho_i]$,

$$<\pi,\sigma> \xrightarrow{\pi} <\rho_i,\sigma>$$

for each $i = 1,\ldots,k$ such that $\sigma(c_i) = T$.

For $\pi = [new\ \bar{y}\ ;\ \rho]$,

$$<\pi,\sigma> \xrightarrow{\pi} <\rho', (\sigma; \bar{y}'{:}\,\bot)>$$

where ρ' and $\bar{y}'$ are obtained from ρ and $\bar{y}$ by systematically renaming all the variables in $\bar{y}$ that are in conflict with the currently declared variables, i.e. the current domain of σ. Again $(\sigma; \bar{y}'{:}\,\bot)$ denotes the state obtained from σ by augmenting the domain of σ by $\bar{y}'$ and assigning to them the undefined value $\bot$.

For $\pi = [\rho_1 \parallel \rho_2]$, we have

$$<\pi,\sigma> \xrightarrow{\lambda} <\rho_1' \parallel \rho_2\ ,\ \sigma'>$$

for each transition $<\rho_1,\sigma> \xrightarrow{\lambda} <\rho_1',\sigma'>$, and

$$<\pi,\sigma> \xrightarrow{\lambda} <\rho_1 \parallel \rho_2'\ ,\ \sigma'>$$

for each transition $<\rho_2,\sigma> \xrightarrow{\lambda} <\rho_2',\sigma'>$

For $\pi = [P\ where\ P \Leftarrow B]$,

$$<\pi,\sigma> \xrightarrow{\pi} <B,\sigma>$$

For $\pi = call\ P$, appearing within the body B of a declaration $P \Leftarrow B$,

$$<\pi,\sigma> \xrightarrow{\pi} <B,\sigma>.$$

If $<\pi,\sigma> \xrightarrow{\lambda} <\pi',\sigma'>$ for some π',σ', then we say that the label (process) λ is ***enabled*** in the configuration $<\pi,\sigma>$.

An execution sequence corresponding to the initial configuration $<\pi_0,\sigma_0>$ is a labelled transition sequence:

$$S: <\pi_0,\sigma_0> \xrightarrow{\lambda_0} <\pi_1,\sigma_1> \xrightarrow{\lambda_1},\ldots$$

such that:

a) Every transition appearing in S is justified by the definition above.

b) The sequence S is ***maximal***, i.e., it is either infinite or terminates in a configuration $<\pi_k,\sigma_k>$ on which no subprocess of π_k is enabled.

c) The sequence S is weakly fair. This means that we exclude infinite sequences in which for some λ and $i \geq 0$, λ is continuously enabled beyond $<\pi_i,\sigma_i>$ but never taken, i.e., λ is enabled in each $<\pi_j,\sigma_j>$, $j > i$, but for all $j > i$, $\lambda_j \neq \lambda$.

We define $S_{\bar{x}}$, the set of ***$\bar{x}$-states***, as the set of all states whose domain is $\bar{x}$. Let $\pi = [own\ \bar{x}; \rho]$ be a complete process, and $s_0 \in S_{\bar{x}}$. A ***behavior*** of π on s_0 is a finite or infinite sequence of $\bar{x}$-states:

$$B: s_0, s_1, \ldots$$

such that there exists an execution sequence:

$$S: <\pi_0,\sigma_0> \xrightarrow{\lambda_0} <\pi_1,\sigma_1> \xrightarrow{\lambda_1},\ldots$$

with $\pi_0 = \pi$ and $s_i = \sigma_i\ |_{\bar{x}}$, i.e., σ_i restricted to the domain $\bar{x}$, for each $i = 0, 1, \ldots$.

This definition of behavior is still too detailed and may contain redundant details such as stuttering. Consequently, we define the notion of a ***reduced behavior*** which eliminates stuttering altogether. A reduced behavior corresponding to a complete process π and an initial state s_0, is a finite or infinite sequence of x-states which is obtainable from a behavior of π on s_0 by deleting all consecutive duplicates. Obviously such a deletion may transform infinite behaviors into finite reduced behaviors. Let $B(\pi, s_0)$ be the set of all reduced behaviors of π on s_0. Then the operational semantics we assign to a complete process π is a mapping from initial states to reduced behaviors given by:

$$O[\![\pi]\!] = \lambda s_0.B(\pi, s_0)$$

This definition leads directly to a definition of an induced observational congruence given by:

The processes π and ρ are ***operationally congruent***, $\pi \sim \rho$ iff for every context $C(\cdot)$

(1) $C(\pi)$ is a well formed complete process iff $C(\rho)$ is.

(2) In the case that both $C(\pi)$ and $C(\rho)$ are well formed complete processes, $O[\![C(\pi)]\!] = O[\![C(\rho)]\!]$.

As an example of this congruence we observe that

$$(rest) \sim [\,\Box F \rightarrow skip] \sim (P\ where\ P \Leftarrow P)$$

We may now reformulate the challenge we posed in the introduction as: Find a compositional semantics which is fully abstract relative to the operational congruence defined above. We claim that the real temporal semantics that we introduce in the next section answers this challenge.

A Real Temporal Semantics

Let $own\ \bar{x}; \pi_0$ be a well formed complete process. Let us associate a temporal proposition variable X_i with each process variable P_i, $i = 1, \ldots k$ defined in π_0. Also assume that we have computed for each subprocess ρ appearing in π_0, its ownership set $owns(\rho)$ determined by its context.

In the section dealing with temporal logic, we have defined the formula $\mathrm{Ch}(\varphi)$ that marks the transition point at which a formula φ changes its truth value. We extend this formula to mark a change in a data variable y by:

$$\mathrm{Ch}(y) = \exists u.\mathrm{Rise}(y = u)$$

This marks the point of a change from $y \neq u$ to $y = u$. We also define the *idling formula* for y:

$$\iota(y) = \neg\mathrm{Ch}(y).$$

The temporal semantics of a process π, denoted by $[\![\pi]\!]$, is a temporal formula that characterizes its behavior in an abstract way. In the following definitions we use the abbreviation $\iota = \iota(owns(\pi))$ to denote that all the variables owned by π are not presently changed. *We provide* one clause of the definition for each type of subprocess:

- $[\![rest]\!] = \iota \wedge \boxplus\iota$
  This implies that the main effect of the process *rest* is to preserve forever the values of variables it owns.
- $[\![call\,P_i]\!] = \iota\mathcal{U}^{+}X_i$
  where X_i is the proposition variable we have associated with the process variable P_i.
- $[\![skip; \rho]\!] = \iota\mathcal{U}^{+}[\![\rho]\!]$
- $[\![y := e; \rho]\!] =$
  $\iota \wedge \exists u[\iota\hat{\mathcal{U}}(\iota \wedge (u = e) \wedge \iota\hat{\mathcal{U}}\{(y = u) \wedge \iota(\mathrm{owns}(\pi) - y) \wedge$
  $\iota\hat{\mathcal{U}}[\![\rho]\!]\}]$

This formula identifies a first point in which e is evaluated, and then a second point at which y is assigned the obtained value while all the other variables owned by π are still preserved.

- $[\![\mathop{\Box}\limits_{j=1}^{k} c_j \to \rho_j]\!] =$

$$\iota \wedge \{[\boxplus\iota \wedge \bigwedge_{j=1}^{k} \boxplus\diamondplus\neg c_j] \vee \bigvee_{j=1}^{k} \iota\hat{\mathcal{U}}[c_j \wedge [\![\rho_j]\!]]\}$$

This definition considers the possibility of deadlock at π if each condition is infinitely many times false. the other possibility is the identification of a true c_j *followed* by the execution of ρ_j.

- $[\![\rho_1 \parallel \rho_2]\!] = [\![\rho_1]\!] \wedge [\![\rho_2]\!]$.

We consider the simplicity of this clause an important feature that may well justify the complexity of the other clauses.

- $[\![new\ \bar{x}; \rho]\!] = \iota \wedge \left(\bigwedge_{y \in \mathrm{owns}(\pi) \cap \bar{x}} \boxplus\iota(y)\right) \wedge \exists\bar{x}(\iota\hat{\mathcal{U}}[\![\rho]\!])$

The main effect of the declaration of new variables is expressed by the existential quantification over the newly introduced variables. A secondary effect is that all the variables that π owns but have been *covered* or redeclared in $\bar{x}$, i.e., variables in $\mathrm{owns}(\pi) \cap \bar{x}$, will never be modified again. This is because any reference made by ρ to one of these variables is interpreted as addressing the newly declared variable of that name.

- $[\![P_i \ where\ P_i \Leftarrow B_i]\!] = \exists q.\nu X_i[\iota\mathcal{U}^{+}(\mathrm{Ch}(q) \wedge \iota\hat{\mathcal{U}}[\![B]\!])]$

In principle, the natural definition we would expect for process recursion is:

$$\nu X.[\iota\mathcal{U}^{+}[\![B]\!]].$$

However, as we explained in the introduction, *if* B contains an *unguarded* path, i.e., a path with no change in the values of variables, from P to *call* P, the maximal fixpoint of the naive equation will include undesirable behaviors. To ensure that all paths to X_i in $[\![B]\!]$ will contain a change, we impose an external clock q which is required to change at least once on each recursion. By existentially quantifying over it, we abstract away any particular features that may be associated with a specific clock.

Because of space limitations we present the main theorem of this paper without a proof. A detailed proof will be contained in a technical report presenting a fuller version of the paper.

Theorem

The real temporal semantics presented in this section is fully abstract with respect to the relation of operational congruence.

TLR As a Working Tool

The complex formulae appearing in the definition of the temporal semantics of processes may have created the impression that TLR is a complicated formalism to work with. This impression is unjustified, and the apparent complexity should be attributed to the *efforts* of constructing a compositional semantics of concurrent processes. In fact, for actual reasoning about programs, TLR is quite comparable to integer-based temporal logic, and the added feature of full abstractness makes it an attractive alternative.

Consider for example the following process:

$$\pi\text{: } own\ x; P\ where\ P \Leftarrow [x := x + 1; P]$$

An obvious property of this process is expressed by the formula $(x \geq 0) \to \boxplus(x \geq 0)$. Let φ denote $x \geq 0$. In the integer-based TL we establish the conditional invariance of φ, i.e., that once it holds it is preserved forever, by showing that all the atomic actions of π preserve φ. Here we do something similar. First, we observe that after some simplifications $[\![\pi]\!] = \Theta$ where

$$\Theta\text{: } \nu X.\exists u[(x = u)\mathcal{U}^{+}(x = u + 1)\mathcal{U}^{+}[(x = u + 1) \wedge X]]$$

We have eliminated in this expression the external clock q, since the process itself guarantees a change on each iteration. This elimination can be formally justified. Obviously Θ satisfies its equation:

1. $\Theta \equiv \exists u[(x = u)\mathcal{U}^{+}(x = u + 1)\mathcal{U}^{+}[(x = u + 1) \wedge \Theta]]$
   From which it is not difficult to establish:
2. $\Theta \wedge \varphi \rightarrow \{\varphi\mathcal{U}^{+}[\mathrm{Ch}(x) \wedge \varphi\mathcal{U}^{+}(\Theta \wedge \varphi)]\}$
   This can be interpreted as showing that $\Theta \wedge \varphi$ satisfies the equation
   $$Y \rightarrow \varphi\mathcal{U}^{+}[\mathrm{Ch}(x) \wedge \varphi\mathcal{U}^{+}Y]$$
   Consequently, using rule R3 and the existential version of R4 we obtain
3. $\Theta \wedge \varphi \rightarrow \exists x.\nu Y[\varphi\mathcal{U}^{+}[\mathrm{Ch}(x) \wedge \varphi\mathcal{U}^{+}Y]]$
   An important theorem of TLR is:
4. $\{\exists x.\nu Y[\varphi\mathcal{U}^{+}[\mathrm{Ch}(x) \wedge \varphi\mathcal{U}^{+}Y]]\} \equiv (\varphi \wedge \boxplus\varphi)$
   We thus obtain
5. $\Theta \wedge \varphi \rightarrow \boxplus\varphi$
   Or equivalently
6. $[\![\pi]\!] \rightarrow [\varphi \rightarrow \boxplus\varphi]$
   Using the notation of [BKP1] this is representable as
7. $[\pi]\{\varphi \rightarrow \boxplus\varphi\}$
   which means that all executions of π satisfy the temporal property $\varphi \rightarrow \boxplus\varphi$.

Since the only step in this proof that depends on the specific π and ρ considered, was the derivation of 2 from 1, we can condense all the others into a derived proof principle.

Let π be a process of the form:

$$\pi\text{: } \mathit{own}\ \bar{y};\ P\ \mathit{where}\ P \Leftarrow B$$

Denote by $[\![B]\!](X)$ the temporal semantics of B, where dependence on the propositional variable X has been made explicit. Then we have the following rule:

$$\frac{\Theta \equiv [\![B]\!](\Theta) \quad \vdash \quad \Theta \wedge \varphi \rightarrow \{\varphi\mathcal{U}^{+}[\mathrm{Ch}(q) \wedge \varphi\mathcal{U}^{+}(\Theta \wedge \varphi)]\}}{[\pi]\{\varphi \rightarrow \boxplus\varphi\}}$$

A slightly more general rule is needed for the case that B is not guarded.

Inspecting the passage from 1 to 2 above, we see that what is needed is establishing that φ is preserved along any computation path in B leading to any *call P* appearing within B. We also observe that it is very similar to the rule PROC handling recursion in [BKP1].

It is clear that many more derived rules of this kind should be developed before we can use TLR with the same ease and convenience now attained in the integer-based TL. However, we do feel confident that such high-level rules can and will be developed.

An Example of Specification and Verification

For a more comprehensive example we consider Peterson algorithm for mutual exclusion ([Pe]).

In a slightly extended version of our programming language, the algorithm can be presented as:

```
    $P$: $\mathit{own}\ y_1, y_2, t, in_1, in_2$;
    $(y_1, y_2, t, in_1, in_2) := (F, F, F, F, F); [\rho_1 \parallel \rho_2]$
where
    $\rho_1$: $\mathit{own}\ y_1, in_1, \downarrow t$ ; $[P_1\ \mathit{where}\ P_1 \Leftarrow B_1]$
    $\rho_2$: $\mathit{own}\ y_2, in_2, \uparrow t$ ; $[P_2\ \mathit{where}\ P_2 \Leftarrow B_2]$
    $B_1$: $[(T \rightarrow \mathit{call}\ P_1)$
          $\square$
          $(T \rightarrow y_1 := T;\ t := F$ ;
                 $[Q_1\ \mathit{where}\ Q_1 \Leftarrow C_1])]$
    $B_2$: $[(T \rightarrow \mathit{call}\ P_2)$
          $\square$
          $(T \rightarrow y_2 := T;\ t := T$ ;
                 $[Q_2\ \mathit{where}\ Q_2 \Leftarrow C_2])]$
    $C_1$: $[(y_2 \wedge \neg t \rightarrow \mathit{call}\ Q_1)$
          $\square$
          $(\neg y_2 \vee t \rightarrow in_1 := T$ ; $in_1 := F$ ;
                 $y_1 := F$ ; $\mathit{call}\ P_1)]$
    $C_2$: $[(y_1 \wedge \neg t \rightarrow \mathit{call}\ Q_2)$
          $\square$
          $(\neg y_1 \vee \neg t \rightarrow in_2 := T$ ; $in_2 := F$ ;
                 $y_2 := F$ ; $\mathit{call}\ P_2)]$
```

The extension we introduced to our programming language is that both ρ_1 and ρ_2 are allowed to modify t, but each in its own way. The notation $\downarrow t$ means that ρ_1 and its subprocesses are only allowed to set t to F, while ρ_2 is only allowed to set t to T.

As a result the ι formula for ρ_1 and ρ_2 should read respectively:

$$\iota_1 = \neg\mathrm{Ch}(y_1) \wedge \neg\mathrm{Ch}(in_1) \wedge \neg\ \mathit{Fall}\ (t)$$
$$\iota_2 = \neg\mathrm{Ch}(y_2) \wedge \neg\mathrm{Ch}(in_2) \wedge \neg\ \mathit{Rise}\ (t)$$

Writing the semantics of the two processes, it is possible to infer from them the following modular specifications:

$$[\rho_1]\{\ \square(in_1 \rightarrow \Theta_1) \wedge \square(in_1 \wedge \Theta_2 \rightarrow t)\}$$
$$[\rho_2]\{\ \square(in_2 \rightarrow \Theta_2) \wedge \square(in_2 \wedge \Theta_2 \rightarrow \neg t)\}$$

where Θ_1 and Θ_2 characterize the history of a point in which ρ_1 and ρ_2 are ready to enter their critical section (signified by setting in_1 and in_2 to T).

$$\Theta_1 : \iota_1 \wedge \iota_1\hat{S}(\neg t \wedge y_1)$$
$$\Theta_2 : \iota_2 \wedge \iota_2\hat{S}(t \wedge y_2)$$

It is easy to see that when we combine these specifications we can obtain (by contradiction):

$$[\rho_1 \| \rho_2] \{ \Box\neg(in_1 \wedge in_2)\}$$

which establishes mutual exclusion.

Conclusions

The real-numbers based model and its associated real temporal logic, seem to achieve a higher degree of abstractness than the one provided by the integers-based model. The price does not appear to be excessive since the basic structure of temporal formulae, specifications and proofs is not significantly altered. The gain is obvious since it provides a much cleaner and more natural semantics. This becomes even more apparent when illustrated on a communication based process language such as CCS. It can be shown that the real temporal semantics of CCS attains the same standard of abstractness set up in the algebraic treatment of CCS and its derivatives ([M2], [HM], [dNH]).

Acknowledgements

We would like to gratefully acknowledge the support given by the Weizmann Institute to the visit of the first two authors. Many thanks are due to L. Lamport, M. Chandi and J. Misra for most illuminating discussions, to A. Emerson and L. Zuck for friendly help and advice, to the participants of E.W. Dijkstra's Tuesday afternoon club for many helpful comments, and last but not least to C. Weintraub for her most speedy and efficient typing.

References

[BKP1] Barringer, H., Kuiper, R., Pnueli, A. — Now You May Compose Temporal Logic Specifications, 16th STOC (1984) 51–63.

[BKP2] Barringer, H., Kuiper, R., Pnueli, A. — A Compositional Temporal Approach to a CSP-like Language, Proc. of IFIP Conference: The Role of Abstract Models in Information Processing, Vienna (1985).

[dBMO] de Bakker, J.W., Meyer, J.-J.Ch., Olderog, E.-R. — Infinite Streams and Finite Observations in the Semantics of Uniform Concurrency, 12th ICALP (1985) 149–157.

[Br] Brookes, S.D. — A Semantics and Proof System for Communicating Processes, 2nd Workshop on Logics of Programs, *LNCS* 164 (1983) 68–85.

[Bu] Burgess, J.P. — Basic Tense Logic, in D. Gabbay and F. Guenthner (eds.) Handbook of Philosophical Logic, Vol II, D. Reidel Publishers (1984) 89–133.

[CE] Clarke, E.M., Emerson, E.A. — Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic, 1st Workshop on Logic of Programs, *LNCS* 131 (1981) 52–71.

[CM] Clarke, E.M., Mishra, B. — Automatic Verification of Asynchronous Circuits, 2nd Workshop on Logics of Programs, *LNCS* 164 (1983) 101-115.

[HM] Hennesy, M.C.B., Milner, R. — Algebraic laws for Nondeterminism and Concurrency, *JACM 32*, 1 (1985) 137–161.

[HO] Hailpern, B., Owicki, S. — Modular Verification of Computer Communication Protocols, *IEEE Trans. on Communications*, COM-31, 1 (1983) 56–68.

[HP] Hennesy, M.C.B., Plotkin, G.D. — Full Abstraction for a Simple Parallel Programming Language, Mathematical Foundations of Computer Science, *LNCS*, 74, Springer Verlag (1979) 108–120.

[J] Jones, C.B. — Software Development: A Rigorous Approach, Prentice Hall International Series in Computer Science.

[La1] Lamport, L. — What Good is Temporal Logic?, Proc. IFIP Congress, Paris (1983) 657–668.

[La2] Lamport, L. — Specifying Concurrent Program Modules, *ACM TOPLAS 5*, 2 (1983) 190–222.

[LP] Lichtenstein, O., Pnueli, A. — A Deductive System for the Temporal Logic of the Reals, Technical Report, Weizmann Institute of Science, in preparation.

[LPZ] Lichtenstein, O., Pnueli, A., Zuck, L. — The Glory of the Past, Logics of Programs, *LNCS*, 193, Springer Verlag (1985) 196–218.

[M1] Milner, R. — Fully Abstract Models of Typed γ-Calculi, Theoretic Computer Science (1977).

[M2] Milner, R. — A Calculus of Communicating Systems, *LNCS* 92 (1980).

[MP1] Manna, Z., Pnueli, A. — Verification of Concurrent Programs: The Temporal Framework, in Correctness Problem in Computer Science, R.S. Boyer, J.S. Moore (eds.) Academic Press (1982) 215–273.

[MP2] Manna, Z., Pnueli, A. — How to Cook a Temporal Proof System for Your Pet Language, 10th POPL (1983) 141–154.

[MP3] Manna, Z., Pnueli, A. — Verification of Concurrent Programs: A Temporal Proof System, Foundations of Computer Science IV, Distributed Systems, *Mathematical Centre Tracts*, 159, Amsterdam (1983) 163–255.

[dNH] de Nicola, R., Hennesy, M.C.B. — Testing Equivalence for Processes, 10th ICALP, *LNCS* 154 (1983).

[NGO] Nguyen, V., Gries, D., Owicki, S. — A Model and Temporal Proof System for Networks of Processes, 12th POPL (1985).

[OL] Owicki, S., Lamport, L. — Proving Liveness Properties of Concurrent Programs, *ACM TOPLAS 4*, 3 (1982) 455–495.

[Pe] Peterson G.L. — Myths about the Mutual Exclusion Problem, *Information Processing Letters* 12,3(1981) 115–116.

[Pn1] Pnueli, A. — The Temporal Semantics of Concurrent Programs, *Theoretical Computer Science* 13 (1981) 45–60.

[Pn2] Pnueli, A. — In Transition from Global to Modular Temporal Reasoning About Programs, Proc of NATO School on Logic and Models for Verification and Specification of Concurrent Systems, La Colle-Sur-Loup (1984).

[SMS] Schwartz, R.L., Melliar-Smith, P.M. — Temporal Logic Specifications of Distributed Systems, 2nd International Conference on Distributed Computing Systems, Paris (1981).

Systolic Algorithms as Programs

K. Mani Chandy
J. Misra

Department of Computer Sciences
University of Texas
Austin, Texas 78712

20 December 1985

Abstract

We represent a systolic algorithm by a program consisting of one multiple assignment statement that captures its operation and data flow. We use invariants to develop such programs systematically. We present two examples, matrix multiplication and LU-decomposition of a matrix.

This work was supported in part by a grant from the Office of Naval Research under grant N00014-85-K-0057.

Table of Contents

List of Figures

1. Introduction

Systolic algorithms [1] are synchronous, parallel programs executing on a number of nodes (machines) interconnected by a set of lines. Systolic algorithms are often described by pictures of nodes and lines and descriptions of processing at each node in the picture and data movement between nodes. A pictorial representation of an algorithm suggests that it can be implemented on a VLSI chip; however, pictures do not lend themselves readily to proofs. of correctness.

We view systolic algorithms as programs and apply traditional program development techniques, based on invariants, in their design. In this paper we carry out the development of algorithms for matrix multiplication of band matrices and L-U decomposition of a band matrices. Both algorithms are from Kung and Leiserson [1].

We are far from proposing a VLSI design methodology: We do not consider many of the limitations in a physical realization; these are concerns for a later stage in the design. However, our use of traditional program development techniques seems to yield designs for which data flow rates, initial and boundary conditions-- the tedious details-- are derived mechanically.

A great deal of work has been done on systematic methods for developing systolic algorithms [5,6,7,8]. These methods are largely based on transforming sets of equations into forms suitable for implementation on systolic hardware. The primary contribution of this paper is to represent systolic algorithms by *programs derived from invariants.* Each program consists of *one* multiple assignment statement. Our goal is to apply traditional programming techniques in developing systolic algorithms.

2. Programs and Systolic Algorithms

2.1. Programs

Our programs have *multiple assignment* statements. A multiple assignment statement of the form,

$$x, y := f(x, y), g(x, y)$$

assigns $f(x', y')$ and $g(x', y')$ to x, y respectively where x', y' are the values of x, y prior to the execution of the statement. We allow the right sides of assignments to be conditional expressions. For instance, we represent

$$x := \begin{cases} 0, & \text{if } a > 0 \\ 1, & \text{if } a \leq 0 \end{cases}$$

by,

$x := 0 \text{ if } a > 0 \sim 1 \text{ if } a \leq 0$

A program consists of *declarations* of its variables and their initial values and *one* multiple assignment statement. The program execution consists of executing this statement repeatedly forever. Non-terminating execution is convenient for reasoning; however, the program may be stopped when the left and right sides of the statement are equal in value, because no further change in variable values is then possible. Restricting a program to one multiple assignment may seem too restrictive. However, our experience suggests that such programs are adequate for representing systolic algorithms. A multiple assignment can be thought of as a synchronous computation -- computing all expressions on the right side synchronously -- and hence, captures the essence of systolic computations. Elsewhere [2,3,4] , we have shown that a *set* of multiple assignment statements executed in a non-deterministic fashion represents different kinds of parallel and distributed computations; for this paper, we do not require this generality.

2.2. Systolic Algorithms

A systolic algorithm is executed on a collection of nodes, and directed lines connecting pairs of nodes. A *step* of the computation consists of some nodes (1) reading values from (some or all of) their input lines, (2) computing and (3) writing values to (some or all of) their output lines. A value written to a line is available at the *next step* at the node to which the line is directed. We may represent local data at a node by placing the data on lines directed from the node to itself.

Systolic algorithms display regular structures: there are only a few kinds of nodes, and interconnections among nodes are regular. Furthermore, in many cases, systolic hardware operates in a pipelined fashion.

2.3. Representing Systolic Algorithms by Programs

We represent each line in a systolic circuit by a variable; a variable value at any point in the computation is the value on the corresponding line. Each node in a systolic circuit is represented by an assignment (in the multiple assignment statement). A synchronous step in the systolic algorithm is simulated by executing a multiple assignment statement: it assigns new values to certain variables based on current values of some variables. A small example is given below.

Example: (Shift Register)

A systolic algorithm for a shift register with N nodes is shown pictorially in figure 2-1. Every node transmits the value from its input line to its output line in every step. Lines are numbered as shown in the picture.

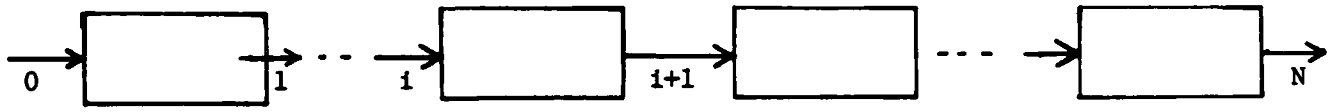


Figure 2-1: Shift Register

Let $x[i]$ be the variable associated with the ith line. The multiple assignment statement which represents the operation of this algorithm is, (informally)

for all i in 0 to $N-1$:: {assign in parallel}
$x[i+1] := x[i]$.

We will write this as (in a notation to be introduced later):

```
⟨i in 0..N−1::
  || x[i + 1] := x[i]
⟩
```

Note that there is no explicit mention in the program about data movement. Data items move within the array by being assigned to different array elements, but our treatment *does not* trace the movement of individual data items.

A multiple assignment statement may represent an algorithm having no systolic realization. For instance, a line value is read by exactly one node in a systolic algorithm but a variable may appear in the right hand side of more than one assignment in our program. Similarly, computation at a node usually depends only on a few input line values due to physical constraints, but our programs allow expressions on the right hand side to have arbitrary numbers of variables. We constrain our programs to mirror these limitations of systolic hardware.

Limited fan-in, fan-out: Each expression on the right hand side of an assignment has a bounded number of variables. This bound is the maximum fan-in. Each variable appears at most once on the left hand side of an assignment and at most once on the right hand side of an assignment; this is because each line is directed from one node on the external environment to one node on the external environment.

Systolic algorithms typically operate on arrays of data items. Systolic algorithms require that the speed at which data moves through the circuit be independent of the index of the data items (usually). Hence, we propose:

Linearity: The step number at which a computation is done is usually a simple (e.g., linear or piecewise-linear) function of data indices.

We have shown the correspondence between systolic algorithms and a special kind of program. Henceforth, we deal only with issues of developing such a program from a specification.

2.4. Program Development

As in other areas of programming, an *invariant* is a central concept in our approach to systolic algorithms. In fact, it seems that the program design task is almost over once a suitable invariant is found. We introduce a variable t, denoting the step number (t is initially 0 and is increased by 1 in each execution of the statement) and state an invariant relating various data items and t. We will be guided by the limited fan-in-fan-out and linearity requirements in postulating an invariant. The effect of statement execution is to preserve the invariant when t is increased by 1.

The invariant is useful in deriving initial conditions and boundary conditions. Determining these conditions and the rate of data flow are the most tedious details one has to contend with; invariants seem to simplify the effort.

2.5. Notation

We use || to break up a multiple assignment statement into its component assignments for convenience in reading. For instance,

$$x, y := y, x$$

is equivalent to

$$x := y \parallel y := x \quad .$$

The following notation, where S is a set and each $Q(i)$ is an assignment (or multiple assignment):

$$\langle\, i \text{ in } S :: \parallel Q(i) \,\rangle$$

denotes a statement obtained by enumerating, for every element of S, $Q(i)$ with i replaced by that element. For example $\langle\, i \text{ in } 0..1 : \parallel X[i] := Y[i] \,\rangle$ is equivalent to $\parallel X[0] := Y[0] \parallel X[1] := Y[1]$. We omit S when it is clear from the context. The statement,

$x := e$ if b

is to be interpreted as

$x := e$ if $b \sim x$ if $\neg b$.

The scope of *if* will be shown explicitly, if needed, as in the following.

$x, y := (e_1, e_2)$ if b

is equivalent to,

$x, y := e_1$ if b, e_2 if b

and also equivalent to,

$(x, y := e_1, e_2)$ if b

3. Band Matrix Multiplication

The problem is to compute

$$C = A \cdot B$$

where A, B are band matrices and "$\cdot$" denotes multiplication.

We have,

$$C[i,k] = \sum_j A[i,j] \times B[j,k]$$

This expression cannot be used directly for computing $C[i,k]$ since that would violate the limited fan-in-fan-out requirement. Therefore, we define as in [1]:

$$C^j[i,k] = \begin{cases} 0, & \text{if } j < 0 \\ C^{j-1}[i,k] + A[i,j] \times B[j,k], & \text{if } j \geq 0 \end{cases} \quad (1)$$

Equation (1) suggests that $A[i,j]$ and $B[j,k]$ will be multiplied in some step. Using the linearity criterion, we may postulate that they will be multiplied in a step which is a linear function of i,j,k. If this linear function is independent of one of its arguments, say i, then for any fixed value of j,k, $A[i,j]$ and $B[j,k]$, will be multiplied in the same step for all i; that is $B[j,k]$ will appear in more than one computation in a step, thus violating the limited fan-in-fan-out requirement. Hence, we may assume that $A[i,j]$, $B[j,k]$ are multiplied in a step that is a nontrivial linear function of each of its arguments - we choose the simplest such function: $i+j+k$.

Since A,B are band matrices, we postulate that each diagonal (main, subdiagonal or superdiagonal) is pipelined. Let one node be assigned for each pair of diagonals - one from A and one from B - to carry out computations on element pairs from these diagonals. Element $A[i,j]$ belongs to diagonal $(i-j)$ of A and $B[j,k]$ to diagonal $(j-k)$ of B; hence index the node at which they are multiplied by $(i-j,\ j-k)$.

Equation (1) suggests that $A[i,j]$, $B[j,k]$, $C^{j-1}[i,k]$ be made available at the same time at some node and, from this discussion, that node is $(i-j,\ j-k)$. Therefore, each node (v,w) has three input lines $X[v,w], Y[v,w]$ and $Z[v,w]$, along which A,B,C respectively are pumped into it. From this discussion, we have the following invariant.

Invariant : $t=i+j+k \Rightarrow$

$$\begin{aligned}[&X[i-j,\ j-k]=A[i,j] \quad \text{and,}\\ &Y[i-j,\ j-k]=B[j,k] \quad \text{and,}\\ &Z[i-j,\ j-k]=C^{j-1}[i,k].\\ ]&\end{aligned}$$

The variables i,j,k,t in the invariant are universally quantified over all integers; ignore the equations corresponding to undefined subscript values in the right side.

Our design task is nearly complete! We merely have to show how to establish the invariant initially, and how to preserve it when t is increased by 1.

3.1. Initial Conditions

Initially, let t be 0. Then for any i,j with $k=-(i+j)$, we are required to have,

$$X[i-j,\ i+2j]=A[i,j].$$

Similarly,

$$Y[-2j-k,\ j-k]=B[j,k].$$

Let $j=-(i+k)$, where $i\geq 0, k\geq 0$. Then, $j\leq 0$. Hence,

$$C^{j-1}[i,k]=0.$$

Substituting $-(i+k)$ for j in the invariant,

$$Z[2i+k,\ -i-2k]=0.$$

Summarizing the initial conditions,

$$\begin{aligned}&X[i-j,\ i+2j]=A[i,j], \text{ for all } i,j\\ &Y[-2j-k,\ j-k]=B[j,k], \text{ for all } j,k\\ &Z[2i+k,\ -i-2k]=0, \text{ for } i\geq 0,\ k\geq 0.\\ &t=0\end{aligned}$$

3.2. Preserving the Invariant

We show how to preserve the invariant when t is increased by 1. First, we simplify the notation by introducing,

$$v=i-j \text{ and } w=j-k.$$

First consider the data item $A[i,j]$. From the invariant, it equals $X[v,w]$ at $t=i+j+k$. It must equal $X[v,\ w-1]$ after t is increased by 1. This can be accomplished by the assignment,

$$X[v,\ w-1]:=X[v,w].$$

Similarly, we get the assignments,

$$\begin{aligned}&Y[v+1,w]:=Y[v,w] \quad \text{and,}\\ &Z[v-1,w+1]:=Z[v,w]+X[v,w]\times Y[v,w].\end{aligned}$$

Note that these steps need be carried out only for t,i,j,k satisfying $t=i+j+k$, i.e., $t=(i-j)-(j-k)+3j$. We rewrite this condition - weakening it somewhat, to eliminate i,j,k - as $t=(v-w)$ mod 3. This results in the following program.

```
Program P1 {for multiplying band matrices}

 initially :

    ⟨ for all $i,j$ :: $X[i-j,\ i+2j]$     $= A[i,j]$ ⟩
    ⟨ for all $j,k$ :: $Y[-2j-k,\ j-k]$    $= B[j,k]$ ⟩
    ⟨ for all $i,k$ :: $Z[2i+k,\ -i-2k]$   $= 0$ ⟩

    assign: ⟨ for all $v, w$::
              ( ||      $X[v,w-1]$  :=  $X[v,w]$
                ||      $Y[v+1,w]$  :=  $Y[v,w]$
                || $Z[v-1,w+1]$     :=  $Z[v,w]+X[v,w]\times Y[v,w]$  )
                                                      if $t=(v-w) \bmod 3$ ⟩
                ||             $t$  :=  $t+1$

end {P1}
```

This program represents a systolic array. We have finished a large part of the design. What remains to be done is to determine the size of the systolic array and the number of steps required to complete execution.

3.3. Determining Array Size and Number of Steps

Program $P1$ does not specify the dimensions of X,Y,Z nor the step number t, up to which program execution should continue. These parameters, and others, can be deduced from the invariant using the sizes of input matrices A,B as parameters.

Let BA,TA (bottom of A, top of A) have the following meaning: $A[i,j]$ is zero unless $BA \le i-j \le TA$. Likewise, define BB,TB for matrix B. The multiplication in program $P1$ yields a zero if $X[v,w]=0$ or $Y[v,w]=0$. Therefore, we may restrict v,w to the range $BA \le v \le TA$ and $BB \le w \le TB$ for computation of the product. Hence, Z can be dimensioned $(BA-1\ ..\ TA,\ BB\ ..\ TB+1)$. Other assignments merely move the elements of A or B; this corresponds to feeding the systolic array appropriate elements of A and B.

Next, we determine when and where $C[i,k]$, for any given i,k, becomes available. That is, we want to find T and v,w such that,

$$(t=T) \Rightarrow (Z[v,w]=C[i,k]).$$

First we determine j such that:

$$C[i,k]=C^{j-1}[i,k]. \qquad (2)$$

This holds when $A[i,j]=0$ or $B[j,k]=0$. ($A[i,j]=0$ and $B[j,k]=0$ if j exceeds the number of columns of A. To eliminate special case analysis , we assume that A,B are augmented with suitable number of zeros for larger values of j.)

$$A[i,j]=0, \text{for } j \geq 0, \text{if } i-j < BA,$$

$$B[j,k]=0, \text{for } j \geq 0, \text{if } j-k > TB.$$

Hence the minimum value of j for which (2) holds is j^* given by

$$j^* = min(i-BA,\ k+TB) \ + \ 1$$

From the invariant, at $T=i+j^*+k$,

$$Z[i-j^*,\ j^*-k]=C[i,k].$$

Note that in case $-BA=TB$, $j^*=min(i,k)+TB+1$. Hence the systolic array has a pleasing diamond structure as given in [1]. However, for arbitrary BA,TB, the structure is not as regular. We show the relevant portion (i.e., where multiplication is done) of an arbitrary systolic array in figure 3-1.

The invariant simplified the considerations of initial and boundary conditions and data flow rates. In this particular example, we imagined that all the elements of matrices A,B are initially placed on certain lines, though the useful work (of multiplication) is performed in a limited region. Now we consider an example, L-U decomposition, where such an assumption cannot be made; in fact, the goal of the algorithm is to compute something akin to A,B from C.

4. L-U Decomposition of a Band Matrix

Given a band matrix A, its L-U decomposition is a pair of matrices L and U where L is a lower diagonal matrix with 1's on diagonals and U is an upper diagonal matrix, satisfying the following equations. (These equations are from [1] with indices renamed and renumbered starting from 0.)

$$A^0[i,k] \quad = \quad A[i,k]$$

$$A^{j+1}[i,k] \quad = \quad A^j[i,k] - L[i,j] \times U[j,k],\ j \geq 0$$

$$L[i,j] \quad = \quad \begin{cases} 0, & \text{if } i < j \\ 1, & \text{if } i = j \\ A^j[i,j]/U[j,j], & \text{if } i > j \end{cases}$$

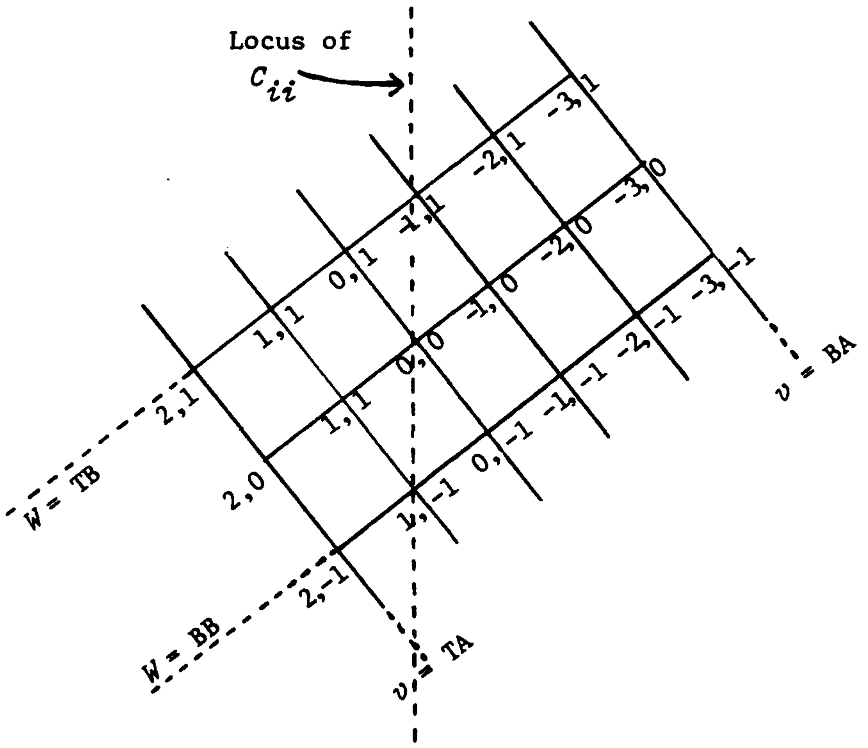


Figure 3-1: Relevant portion of a systolic array for multiplications of band matrices with $BA = -3$, $TA = 2$, $BB = -1$, $TB = 1$

$$U[j,k] = \begin{cases} 0, & \text{if } j > k \\ A^j[j,k], & \text{if } j \leq k \end{cases}$$

We adopt the convention that $A^j[i,k] = A[i,k]$, for $j \leq 0$. As described in the last section, let BA,TA be such that $A[i,j] = 0$ unless $BA \leq i - j \leq TA$. It can be shown for band matrices that

$$A^{j+1}[i,k] = \begin{cases} A[i,k], & \text{if } (i-j > TA) \text{ or } (j-k < BA) \\ A^j[i,k] - L[i,j] \times U[j,k], & \text{otherwise} \end{cases}$$

The form of computation on A suggests matrix multiplication. Hence, we attempt using the invariant for matrix multiplication, with variables suitably renamed for this problem. In the following invariant we have constrained certain indices because L is a lower diagonal and U an upper diagonal matrix.

Invariant:

$t = i + j + k \Rightarrow$

$[(j \geq 0, i \geq j, k > j \quad \Rightarrow X[i-j, j-k] = L[i,j])$ and

$(j \geq 0, i > j, k \geq j \quad \Rightarrow Y[i-j, j-k] = U[j,k])$ and

$(i \geq 0, k \geq 0, i \geq j, k \geq j \quad \Rightarrow Z[i-j, j-k] = A^j[i,k])$

$]$

As before, we give initial conditions satisfying the invariant and show how to preserve the invariant when t is increased by 1. The major difference from matrix multiplication is that L,U, unlike matrix multiplication, are not available initially and have to be computed.

4.1. Initial Conditions

For $t = 0$, the first two conditions in the invariant are vacuously satisfied because there are no i,j,k satisfying these conditions. The last condition, for any $i \geq 0$, $k \geq 0$, can be satisfied by letting $j = -(i+k)$ and hence,

$$Z[2i+k, \ -i-2k] = A^j[i,k] = A[i,k] \quad \{\text{since } j \leq 0\}$$

4.2. Preserving the Invariant

As before, we use

$$v = i - j \quad \text{and} \quad w = j - k.$$

It follows from the invariant that we need only consider the cases for $v \geq 0$ and $w \leq 0$.

4.2.1. Preserving the first condition in the invariant

We now consider preservation of,

$t = i + j + k$ and $j \geq 0, i \geq j, k > j \Rightarrow$
$X[v,w] = L[i,j]$.

For any i,j,t where $i \geq j \geq 0$ and $t > i + 2j$: there is some (v,w), $v \geq 0$, $w \leq 0$ such that, $X[v,w] = L[i,j]$.

We now ask ourselves how this requirement is to be met. If $t > i+2j$, $L[i,j]$ has already been computed and has to be assigned to the proper $X[v,w]$. If $t = i+2j$, then $L[i,j]$ is to be computed and assigned to the appropriate $X[v,w]$.

Case 1) $t > i + 2j$ {equivalently, $w < 0$}:

Then, $X[v,w] = L[i,j]$.

The invariant is preserved by:

$$X[v, w-1] := X[v,w]$$

Case 2) $t = i+2j$ {equivalently, $w = 0$}:

The invariant is preserved by computing $L[i,j]$ according to its defining equation and then assigning it to the proper variable.

At the following step, i.e., the $(t+1)^{\text{th}}$ step, the only k satisfying

$$t+1 \;=\; i+j+k$$

is $k \;=\; j+1$.

Hence at this step we must have

$$X[v,-1] = L[i,j]$$

substituting for $L[i,j]$:

$$X[v,-1] := \{L[i,j] = \}A^j[i,j]/U[j,j] \quad \text{if } i > j \quad \sim \quad 1 \quad \text{if } i = j.$$

From the invariant, at $t = i+2j$, $A^j[i,j] = Z[i-j,0]$.

Also, at $t = i+2j$ $\{i > j,\ k = j\}$, $U[j,j] = Y[i-j,0]$.

Hence, the following assignment preserves the invariant {$i > j$ is rewritten as $v > 0$}:

$$X[v,-1] := Z[v,0]/Y[v,0] \quad \text{if } v > 0 \quad \sim \quad 1 \quad \text{if } v = 0.$$

4.2.2. Preserving the second condition in the invariant

By similar reasoning we identify two cases.

Case 1) $t > 2j + k$ {equivalently $v > 0$}:

$$Y[v+1, w] := Y[v,w]$$

Case 2) $t = 2j + k$ {equivalently $v = 0$}:

$$Y[1,w] := A^j[j,k]$$

At $t = 2j + k$, $A^j[j,k] = Z[0, j-k]$. Hence,

$$Y[1,w] := Z[0,w]$$

4.2.3. Preserving the third condition in the invariant

From the equations,

$$A^{j+1}[i,k] = \begin{cases} A[i,k], \text{ if } i - j > TA \text{ or } j - k < BA \\ A^j[i,k] - L[i,j] \times U[j,k], \text{ otherwise} \end{cases}$$

By similar arguments we derive the following assignment to preserve the invariant.

$$Z[v-1, w+1] := \begin{cases} Z[v,w] \quad \text{if } v > TA \text{ or } w < BA \\ Z[v,w] - X[v,w] \times Y[v,w] \quad \text{if } v \leq TA \text{ and } w \geq BA \end{cases}$$

Program $P2$ {L-U decomposition of a band matrix}

```
  initially : $t = 0$,
              ⟨for all $i \geq 0, k \geq 0$ : $Z[2i+k, -i-2k] = A[i,k]$ ⟩

  assign    : ⟨ for all $v,w$::

  ( ||      $X[v, w-1]$ := $X[v,w]$ if $w < 0 \sim Z[v,0]/Y[v,0]$ if $w = 0$ and $v > 0$
                           $\sim$ 1 if $w = 0$ and $v = 0$

    ||      $Y[v+1, w]$ := $Y[v,w]$ if $v > 0 \sim Z[0,w]$ if $v = 0$

    ||   $Z[v-1, w+1]$ := $Z[v,w] - X[v,w] \times Y[v,w]$ if $v \leq TA$ and $w \geq BA$
                           $\sim$ $Z[v,w]$ if $v > TA$ or $w < BA$ )

                                                       if $t = (v-w)$ mod 3 ⟩

    ||              $t$ := $t+1$

end {$P2$}
```

5. Discussion

Programs $P1$, $P2$ capture the essence of the algorithms given in [1] for the corresponding problems. The multiple assignment statement in each program can be implemented by associating a node with each (v,w) and carrying out the operations in each step as given by the algorithm. The algorithm tells us that node v,w accepts inputs along $X[v,w]$, $Y[v,w]$, $Z[v,w]$ and produces results along $X[v,w-1]$, $Y[v+1,w]$ and $Z[v-1,w+1]$ in each step t, where $t = (v-w)$ mod 3. Some ingenious optimizations have been applied in [1] so that only one kind of node - that receives three values A,B,C and computes $A + B \times C$ - may be used almost completely throughout the systolic array.

This work is part of an ongoing project called UNITY [2,3,4] to provide a unified framework for the development of sequential, parallel and distributed programs. A thesis of UNITY is that early stages of program design should not be concerned with architectural and programming language issues: these concerns are appropriate only for later stages of design. Another thesis is that diverse applications - ranging from VLSI algorithms to communication protocols, from command and control systems to spreadsheets - are *programs and amenable to a common design strategy.* UNITY has yet to give conclusive proof of these theses - but we are hopeful.

VLSI implementations require considerably more than an algorithmic description. We have only addressed concerns dealing with correctness arguments and systematic program development. It is straight-forward to map our programs to circuits with limited fan-in and where each line is directed from one node to one node. However, not all such circuits can be implemented in VLSI.

6. References

1. H. T. Kung and Charles E. Leiserson, "Algorithms for VLSI Processor Arrays," (Section 8.3) *Introduction to VLSI Systems*, (ed. Mead and Conway), Addison-Wesley Publishing Company, 1980.

2. K. M. Chandy, "Concurrency for the Masses", Invited Address: Third Annual ACM Symposium on Principles of Distributed Computing, August 1984, Vancouver, Canada, (appeared in *Proceedings of Fourth Annual ACM Symposium on Principles of Distributed Computing, 1985.*

3. K. M. Chandy and J. Misra, "An Example of Stepwise Refinement of Distributed Programs: Quiescence Detection," to appear in *ACM Transactions on Programming Languages and Systems.*

4. K. M. Chandy and J. Misra, "Programming and Parallelism: The Proper Perspective", Research Report, Computer Sciences Department, University of Texas, November 1985.

5. C. Leiserson, F. Rose, and J. Saxe, "Optimizing Synchronous Circuitry by Retiming", Third Caltech Conference on VLSI, (ed. R. Bryant), California Institute of Technology, March 1983, pp. 87-116.

6. G. J. Li, And B. W. Wah, "The Design of Optimal Systolic Arrays", *IEEE Transactions on Computers, C-34*, No. 1, January 1985, pp. 66-77.

7. Marina C. Chen, "A Parallel Language and its Compilation to Multiprocessor Machines or VLSI", Research Report, Yale University, DCS-RR-432, October 1985.

8. Marina C. Chen, "The Generation of a Class of Multipliers: A Synthesis Approach to the Design of Highly Parallel Algorithms in VLSI", Research Report, Yale University, DCS-RR-442, December 1985.

END

DTIC

5-86